**Міністерство освіти і науки України**
**Тернопільський національний технічний університет імені Івана Пулюя**

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету )
Кафедра комп'ютерних наук
(повна назва кафедри)

# КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: **Розробка програмного забезпечення для опрацювання даних з IoT-сенсорів у розумному середовищі**

Виконав: студент IV курсу, групи СНз-41
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)

_____ Дідух М.П.
(підпис) (прізвище та ініціали)

Керівник _____ Литвиненко Я.В.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Липак Г. І.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Боднарчук І.О.
(підпис) (прізвище та ініціали)

Рецензент _____ Цуприк Г.Б.
(підпис) (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
**Тернопільський національний технічний університет імені Івана Пулюя**

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«_8_» _червня_ 2026 р.

# ЗАВДАННЯ
## НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)
за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)
Студенту Дідуху Мар`яну Петровичу
(прізвище, ім'я, по батькові)
1. Тема роботи Розробка програмного забезпечення для опрацювання даних з IoT-сенсорів у розумному середовищі

Керівник роботи Литвиненко Ярослав Володимирович, доктор технічних наук, професор
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
Затверджені наказом ректора від «*14*» *травня* 2026 року № *4/9-238*
2. Термін подання студентом завершеної роботи 16 червня 2026 р.
3. Вихідні дані до роботи вимоги до системи опрацювання сенсорних даних; типи показників розумного середовища; протоколи передавання даних (MQTT, HTTP); інструментальні засоби — Python 3.12, FastAPI, SQLite.
4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. Розділ 1. Аналіз предметної області та огляд існуючих рішень. 1.1. Інтернет речей і концепція розумного середовища. 1.2. Архітектура IoT-систем та та рівні опрацювання даних.1.3. Протоколи передавання даних. 1.4. Проблеми опрацювання сенсорних даних. 1.5. Огляд існуючих рішень. 1.6. Постановка задачі. Розділ 2. Проєктування системи. 2.1. Вимоги до системи. 2.2. Загальна архутектура. 2.3. Модель даних. 2.4. Конвеєр опрацюваня даних.2.5. Методи виявлення аномалій. 2.6. Обгрунтування вибору технологій. Розділ. 3. Програмна реалізація та тестування. 3.1. Середовище та структура проєкту. 3.2. Моделі даних і валідація. 3.3. Реалізація виявлення аномалій. 3.4. Сховище та агрегація. 3.5. REST-інтерфейс і веб-панель. 3.6. Симулятор сенсорів і приймання через MQTT. 3.7. Тетсування. 3.8. Результати експлуатації. 4. Безпека життєдіяльності, основи охорони праці. Висновки. Перелік джерел. Додатки.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| Безпека життєдіяльності, основи охорони праці | Мариненко С.Ю. кандидат технічних наук, доцент кафедри МТ | | |

7. Дата видачі завдання 26 січня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів роботи | Термін виконання етапів роботи | Примітка |
|---|---|---|---|
| 1. | Ознайомлення з завданням до кваліфікаційної роботи | 26.01.2026 | *Виконано* |
| 2. | Підбір та опрацювання літературних джерел по темі кваліфікаційної роботи | 27.01.2026-16.02.2026 | *Виконано* |
| | | | |
| 3. | Виконання дослідження щодо наявності програмного забезпечення для опрацювання даних з IoT-сенсорами в розумному середовищі | 17.02.2026-10.05.2026 | *Виконано* |
| | Розроблення програмного забезпечення для опрацювання даних з IoT-сенсорами в розумному середовищі | | |
| | | | |
| 4. | Оформлення розділу «Аналіз предметної області та | 11.05.2026-17.05.2026 | *Виконано* |
| | огляд існуючих рішень» | | |
| 5. | Оформлення розділу «Проєктування системи» | 18.05.2026-24.05.2026 | *Виконано* |
| 6. | Оформлення розділу «Програмна реалізація та | 25.05.2026-31.05.2026 | *Виконано* |
| | тестування» | | |
| 7. | Виконання завдання до підрозділу «Безпека життєдіяльності» | 01.06.2026-08.06.2026 | *Виконано* |
| 8. | Виконання завдання до підрозділу «Основи охорони | 01.06.2026-08.06.2026 | *Виконано* |
| | праці» | | |
| 9. | Оформлення кваліфікаційної роботи | 09.06.2026-11.06.2026 | *Виконано* |
| 10. | Нормоконтроль | 12.06.2026-15.06.2026 | *Виконано* |
| 11. | Перевірка на плагіат | 16.06.2026 | *Виконано* |
| 12. | Попередній захист кваліфікаційної роботи | 17.06.2026 | *Виконано* |
| 13. | Захист кваліфікаційної роботи | 18.06.2026 | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

Студент _____ Дідух М.П.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Литвиненко Я.В.
(підпис) (прізвище та ініціали)

# АНОТАЦІЯ

Розробка програмного забезпечення для опрацювання даних з IoT-сенсорів у розумному середовищі // Кваліфікаційна робота освітнього ступеня «Бакалавр» // Дідух Мар`ян Петрович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СНз-41 // Тернопіль, 2026 // С. , рис. – 5 , табл. – 5 , кресл. – , додат. – 2 , бібліогр. – 31.

**Ключові слова:** інтернет речей, IoT-сенсори, розумне середовище, опрацювання даних, виявлення аномалій, mqtt, fastapi, sqlite, потокова обробка.

Кваліфікаційна робота присвячена проєктуванню та реалізації програмного забезпечення для приймання, валідації, аналізу та збереження даних, що надходять від сенсорів розумного середовища. В першому розділі кваліфікаційної роботи розглянуто архітектуру систем Інтернету речей, протоколи передавання даних і типові проблеми опрацювання сенсорних потоків: пропуски, викиди, неузгодженість одиниць вимірювання та апаратні збої давачів.

У другому розділі кваліфікаційної роботи спроєктовано трирівневу модульну архітектуру (ядро правил і обробки, сховище, сервіс подання) та реалізовано наскрізний конвеєр опрацювання: валідація й нормалізація → виявлення аномалій → агрегація → збереження. Виявлення аномалій виконується трьома взаємодоповнювальними методами: контролем фізичних і комфортних діапазонів, ковзним z-показником та контролем швидкості зміни значення.

У третьому розділі реалізовано програмне забезпечення мовою Python із застосуванням фреймворку FastAPI; передбачено приймання даних за протоколом MQTT, вбудований симулятор сенсорів для демонстрації без апаратного забезпечення, збереження у вбудованій базі даних SQLite та вебпанель моніторингу. Коректність роботи підтверджено комплектом автоматизованих тестів та експериментальною експлуатацією системи. Робота

містить відтворюваний програмний код, інструкцію з розгортання та результати оцінювання.

**Об’єкт дослідження** — процес опрацювання даних, що надходять від сенсорів розумного середовища.

**Предмет дослідження** — методи та програмні засоби валідації, виявлення аномалій, агрегації та збереження сенсорних даних.

# ANNOTATION

Development of Software for Data Processing from IoT Sensors in a Smart Environment // Qualification work of the educational level «Bachelor» // Didukh Marian // Ternopil Ivan Puluj National Technical University, Faculty of Computer Sciences Department, group SNz-41 // Ternopil, 20262026 // P. , fig. – 5, tabl. – 5, chair. – , annexes. – 2, references – 31.

**Keywords:** internet of things, iot sensors, smart environment, data processing, anomaly detection, mqtt, fastapi, sqlite, stream processing.

The thesis addresses the design and implementation of software for ingesting, validating, analyzing and storing data produced by sensors of a smart environment. It reviews the architecture of Internet-of-Things systems, data-transmission protocols and the typical problems of processing sensor streams: missing values, outliers, inconsistent measurement units and device faults.

A three-layer modular architecture (processing core, storage, presentation service) is designed, together with an end-to-end pipeline: validation and normalization → anomaly detection → aggregation → persistence. Anomaly detection combines three complementary methods: physical/comfort range checks, a rolling z-score, and a rate-of-change check.

The software is implemented in Python with the FastAPI framework; it supports MQTT ingestion, an embedded sensor simulator for hardware-free demonstration, persistence in an embedded SQLite database, and a monitoring web dashboard. Correctness is confirmed by a suite of automated tests and by experimental operation. The work includes a reproducible codebase, deployment instructions and evaluation results.

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

**IoT (IoT)**– Інтернет речей (Internet of Things)

**API**– програмний інтерфейс застосунку (Application Programming Interface)

**REST**– передавання репрезентативного стану (Representational State Transfer)

**MQTT**– телеметричний транспорт черги повідомлень (Message Queuing Telemetry Transport)

**HTTP**– протокол передавання гіпертексту (HyperText Transfer Protocol)

**JSON**– об’єктна нотація JavaScript (JavaScript Object Notation)

**SQL**– мова структурованих запитів (Structured Query Language)

**OOV**– поза межами словника / нештатне значення (out-of-vocabulary)

**CRUD**– створення, читання, оновлення, видалення (Create, Read, Update, Delete)

**ppm**– частин на мільйон (parts per million)

# ЗМІСТ

# ВСТУП

**Актуальність теми.** Інтернет речей (IoT) перетворився з набору окремих давачів на цілісну екосистему взаємопов’язаних пристроїв, які безперервно генерують дані про стан фізичного середовища. Розумне середовище — житло, офіс, навчальна аудиторія чи виробниче приміщення — є простором щільної сенсоризації, де десятки давачів вимірюють температуру, вологість, концентрацію вуглекислого газу, освітленість, рух і споживану потужність. Цінність таких систем визначається не самим фактом збору даних, а здатністю надійно їх опрацьовувати: відсівати помилкові вимірювання, виявляти відхилення від норми та подавати інформацію у придатному для прийняття рішень вигляді.

Сенсорні дані за своєю природою «зашумлені»: давачі виходять з ладу, втрачають живлення, надсилають значення у різних одиницях або з пропущеними мітками часу. Без проміжного шару опрацювання такі дані не придатні для аналітики й автоматизації. Сучасні засоби розроблення програмного забезпечення — легкі вебфреймворки, протоколи обміну повідомленнями та вбудовані бази даних — дають змогу побудувати прозоре, відтворюване й невибагливе до ресурсів рішення, придатне для роботи на одному вузлі (граничному шлюзі) розумного середовища. В цьому і полягає актуальність теми.

**Мета і завдання дослідження.** Метою роботи є проєктування та програмна реалізація модульного програмного забезпечення для опрацювання даних з IoT-сенсорів у розумному середовищі. Для досягнення мети поставлено такі завдання:

1) проаналізувати архітектуру систем Інтернету речей, протоколи передавання даних і проблеми опрацювання сенсорних потоків;
2) здійснити огляд наявних рішень та обґрунтувати вимоги до системи;
3) спроєктувати архітектуру системи, модель даних і конвеєр опрацювання;
4) обґрунтувати й реалізувати методи валідації даних та виявлення аномалій;

5) реалізувати програмне забезпечення з REST-інтерфейсом, вебпанеллю та прийманням даних через MQTT;
6) розробити симулятор сенсорів для демонстрації й тестування;
7) перевірити коректність роботи засобами автоматизованого тестування та експериментальної експлуатації.

**Методи дослідження.** У роботі використано методи системного аналізу, об’єктно-орієнтованого та компонентного проєктування програмного забезпечення, методи математичної статистики (оцінювання середнього та стандартного відхилення для виявлення викидів), а також методи модульного й інтеграційного тестування.

**Практичне значення одержаних результатів.** Розроблене програмне забезпечення є придатним для практичного застосування у системах моніторингу розумного житла, навчальних і дослідницьких стендах та як основа для подальшого розвитку. Воно є відкритим, відтворюваним і не потребує спеціалізованого апаратного забезпечення завдяки вбудованому симулятору сенсорів.

**Структура роботи.** Робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел і додатків. У першому розділі проаналізовано предметну область. У другому — спроєктовано архітектуру та методи. У третьому — описано програмну реалізацію й результати тестування. Четвертий розділ присвячено безпеці життєдіяльності та основам охорони праці.

# РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

## 1.1. Інтернет речей і концепція розумного середовища

**IoT** (скорочення від англ. *Internet of Things*) — це концепція, яка перекладається як Інтернет речей. Інтернет речей — це мережа фізичних об’єктів (пристроїв, техніки, датчиків), оснащених давачами, обчислювальними модулями та засобами зв’язку, які обмінюються даними між собою та із зовнішніми системами без безпосереднього втручання людини. Термін «Інтернет речей» був запропонований британським винахідником і підприємцем Кевіном Ештоном у 1999 році. Він уперше використав цю назву під час презентації для компанії Procter & Gamble. Ештон запропонував концепцію системи, де фізичні об’єкти (речі) оснащуються мініатюрними сенсорами та підключаються до мережі, що дозволяє їм збирати дані, обмінюватися ними та взаємодіяти без участі людини. Масово вживати цей термін почали з розвитком енергоефективних бездротових технологій та хмарних обчислень.[1]

Розумне середовище є окремим випадком застосування IoT, у якому сенсоризований простір (передусім приміщення) автоматично адаптується до потреб користувача. Типовими показниками, що вимірюються в розумному житлі, є температура повітря, відносна вологість, концентрація вуглекислого газу як індикатор якості повітря, рівень освітленості, наявність руху (присутність) та спожита електрична потужність. На основі цих даних система може керувати опаленням, вентиляцією, освітленням і сигналізувати про небезпечні стани.[1, 2]

Ключовою особливістю розумного середовища є щільність сенсоризації: навіть невелике помешкання може містити десятки давачів, кожен з яких періодично надсилає вимірювання. Унаслідок цього обсяг даних швидко зростає, а вимоги до їхнього опрацювання — валідації, аналізу та збереження — стають визначальними для практичної цінності системи.[3]

Сучасне розумне середовище включає широкий спектр пристроїв, які можна класифікувати за кількома ознаками: функціональним призначенням, типом вимірюваного параметра, способом підключення та рівнем енергоспоживання [1]. Розуміння цієї класифікації є необхідним для проєктування системи опрацювання даних, оскільки різні типи пристроїв генерують потоки даних з різними характеристиками.

За функціональним призначенням пристрої розумного середовища поділяють на три основні категорії. Давачі (сенсори) виконують вимірювання фізичних параметрів середовища і перетворюють їх на цифровий сигнал. Виконавчі механізми (актуатори) отримують керувальні сигнали і виконують фізичні дії — вмикають освітлення, регулюють температуру тощо. Шлюзи та контролери агрегують дані від множини давачів і забезпечують їх передавання до системи опрацювання [2].

За типом вимірюваного параметра давачі розумного середовища охоплюють: температурні (терморезистори NTC/PTC, термопари, цифрові давачі DS18B20); вологісні (ємнісні та резистивні датчики DHT11/DHT22, SHT30); давачі якості повітря (електрохімічні та недисперсійні інфрачервоні (NDIR) давачі $CO_2$ серії MH-Z); фотодатчики освітленості (BH1750, TSL2591); давачі руху на основі пасивного інфрачервоного (PIR) виявлення або ультразвукового зондування [4]. Температурні давачі: точність ±0.1–0.5°C, час відгуку 1–30 с, інтерфейс 1-Wire або $I^2C$. Давачі вологості: відносна похибка 2–5%, чутливість до конденсату. Давачі $CO_2$ (NDIR): діапазон 400–5000 ppm, необхідна калібрування кожні 6–12 місяців. Давачі освітленості: діапазон 1–65535 лк, адаптація до спектру денного/штучного світла. PIR-давачі руху: кут огляду 110–120°, дальність 5–7 м, хибні спрацьовування від домашніх тварин.

За способом підключення розрізняють провідні та бездротові давачі. Бездротові є переважаючими у розумному середовищі завдяки зручності встановлення; найпоширенішими стандартами є Wi-Fi (802.11b/g/n), Bluetooth Low Energy (BLE 5.0), Zigbee (802.15.4) та Z-Wave. Кожен з них має власні характеристики дальності, пропускної здатності та енергоспоживання. Зокрема,

Zigbee та Z-Wave оптимізовані для низькоенергетичних мереш з топологією «сітка», де кожен пристрій є ретранслятором [4].

За рівнем енергоспоживання давачі поділяють на пристрої з живленням від мережі (реле, розетки зі вбудованим лічильником), акумуляторні (PIR-давачі, термостати, від 1 до 5 років від однієї батареї) та пристрої з автономним живленням від сонячних панелей (зовнішні давачі погоди). Це безпосередньо впливає на частоту та затримку передавання даних, що є критичним для системи опрацювання.

Відсутність єдиного стандарту є однією з головних проблем у розвитку екосистеми Інтернету речей. Різні виробники застосовують несумісні протоколи та формати даних, що ускладнює інтеграцію пристроїв різних виробників в єдину систему [3]. У відповідь на цю проблему виникло кілька ініціатив щодо стандартизації.

Консорціум Matter (раніше Project CHIP), підтримуваний Apple, Google, Amazon та Samsung, розробив відкритий стандарт для пристроїв розумного дому. Стандарт Matter 1.3 (2024) підтримує до 128 одночасних пристроїв у локальній мережі, використовує IPv6 та Thread для бездротової mesh-мережі і гарантує наскрізне шифрування всіх команд. Для промислового IoT широко застосовуються OPC UA (IEC 62541) та протокол Modbus [3, 4].

Питання безпеки є критичним для IoT-систем у розумному середовищі. Основні загрози включають несанкціонований доступ до пристроїв через невідомі або незмінені паролі за замовчуванням; перехоплення незашифрованого трафіку MQTT (без TLS); атаки типу «відмова в обслуговуванні» на MQTT-брокер; фізичний доступ до пристроїв та підміну прошивки. OWASP IoT Top 10 визначає ненадійні паролі та відсутність шифрування як дві найкритичніші вразливості [5].

Для захисту системи опрацювання даних розумного середовища рекомендується: застосовувати автентифікацію MQTT за допомогою унікальних логіна/пароля або сертифікатів X.509; шифрувати трафік за допомогою TLS 1.2 або 1.3; обмежувати доступ до брокера списком дозволених IP-адрес; регулярно

оновлювати прошивку пристроїв; вести журнали підключень для виявлення аномальних сесій. Розроблене в межах цієї роботи програмне забезпечення підтримує локальне автономне розгортання, що мінімізує поверхню атаки [5, 6].

Сумісність на рівні даних забезпечується через використання стандартизованих форматів передавання. Найпоширенішим є JSON, який, попри більші накладні витрати порівняно з бінарними форматами, забезпечує легку читабельність та підтримку в усіх мовах програмування. Альтернативи — CBOR (Concise Binary Object Representation, RFC 7049), Protocol Buffers та Apache Avro — дають змогу зменшити обсяг пакетів на 30–60 %, що критично для давачів з обмеженою пропускною здатністю [4].

Ринок розумного дому демонструє стійке зростання: за прогнозами аналітиків, до 2028 року кількість підключених пристроїв у світі перевищить 25 мільярдів, а ринковий обсяг досягне 170 млрд доларів США [1]. Серед ключових тенденцій виділяють кілька напрямів.

Граничні обчислення (edge computing) переміщують аналіз даних ближче до джерела — безпосередньо на шлюз або навіть на мікроконтролер давача. Це дозволяє скоротити затримку реакції з сотень мілісекунд (хмарна обробка) до одиниць мілісекунд, знизити споживання трафіку та підвищити надійність роботи за відсутності з'єднання з інтернетом. Технологія TinyML дає змогу запускати спрощені моделі машинного навчання безпосередньо на мікроконтролерах з оперативною пам'яттю 256 кБ [17].

Штучний інтелект та машинне навчання все більше інтегруються у системи розумного середовища. Алгоритми класифікації аномалій, прогнозування споживання енергії та оптимізації мікроклімату замінюють жорстко закодовані правила, адаптуючись до індивідуальних звичок мешканців. Разом з тим, застосування ШІ ускладнює пояснюваність (explainability) рішень — критичний аспект для систем, що впливають на умови проживання [12, 13].

Проєктована в цій роботі система орієнтується на прозоре та відтворюване опрацювання даних без застосування «чорних ящиків» машинного навчання, що відповідає принципам граничних обчислень і забезпечує роботу в умовах

обмежених обчислювальних ресурсів. Разом з тим, модульна архітектура передбачає можливість інтеграції компонентів ШІ у майбутніх версіях як додаткових модулів конвеєра опрацювання.

### 1.2. Архітектура IoT-систем та рівні опрацювання даних

Попри різноманіття реалізацій, в архітектурі IoT-систем традиційно виокремлюють кілька рівнів. Граничний рівень (edge) утворюють самі давачі та виконавчі механізми разом з інфраструктурою їх під’єднання. Дані з граничного рівня надходять до рівня приймання та опрацювання подій, де відбувається їх валідація, аналіз і збереження. Верхній рівень становлять засоби подання та прийняття рішень: інтерфейси користувача, системи сповіщень і автоматизації.[3, 4]

Опрацювання сенсорних даних доцільно організовувати у вигляді конвеєра (pipeline) — послідовності етапів, кожен з яких виконує одну чітко визначену функцію. Такий підхід забезпечує модульність, тестованість і можливість незалежного вдосконалення окремих етапів. У межах цієї роботи конвеєр складається з валідації та нормалізації, виявлення аномалій, агрегації й збереження.

Важливим архітектурним рішенням є вибір місця обчислень. Опрацювання безпосередньо на граничному вузлі (граничні обчислення, edge computing) зменшує затримки, знижує навантаження на канали зв’язку та підвищує приватність, оскільки дані не залишають межі приміщення. Саме на таку модель — автономний однонодовий шлюз — орієнтоване розроблене програмне забезпечення.[17]

### 1.3. Протоколи передавання даних

Для передавання даних від давачів застосовують різні протоколи, найпоширенішим з яких в IoT є MQTT (Message Queuing Telemetry Transport) —

легкий протокол публікації та підписки (publish/subscribe) поверх TCP. Його перевагами є малі накладні витрати, придатність для пристроїв з обмеженими ресурсами та модель тем (topics), що природно відображає структуру розумного середовища (наприклад, smartenv/<кімната>/<давач>).[5, 6]

Поряд з MQTT широко використовують протокол HTTP та архітектурний стиль REST, зручні для інтеграції з вебзастосунками та для надсилання поодиноких вимірювань. Існують також спеціалізовані протоколи (наприклад, CoAP) для дуже обмежених пристроїв. Для забезпечення гнучкості розроблене програмне забезпечення підтримує приймання даних як через MQTT, так і через HTTP REST, а також через вбудований симулятор.[7]

Формат корисного навантаження у сучасних IoT-системах найчастіше є текстовим у нотації JSON, що спрощує налагодження та інтеграцію. У цій роботі сенсорне повідомлення є об’єктом JSON з полями ідентифікатора давача, типу показника, значення, одиниці вимірювання, приміщення та мітки часу. Порівняння основних протоколів передавання даних наведено у таблиці 1.1.[4]

Таблиця 1.1 — Порівняння протоколів передавання даних в IoT

| **Протокол** | **Модель обміну** | **Переваги** | **Обмеження** |
|---|---|---|---|
| MQTT | публікація/підписка | малі накладні витрати, теми, QoS | потребує брокера |
| HTTP/REST | запит/відповідь | простота, інтеграція з вебзастосунками | більші накладні витрати |
| CoAP | запит/відповідь (UDP) | для дуже обмежених пристроїв | менша надійність (UDP) |

## 1.4. Проблеми опрацювання сенсорних даних

Дані реальних давачів рідко бувають ідеальними. До типових проблем належать: пропущені або несвоєчасні мітки часу; значення, надіслані у вигляді рядка замість числа; неузгодженість одиниць вимірювання (наприклад, температура у градусах Фаренгейта замість Цельсія); фізично неможливі

значення внаслідок несправності давача; різкі викиди та «застрягання» давача на сталому значенні.[12]

Якщо ці проблеми не усунути на ранньому етапі, вони спотворюють подальшу аналітику та можуть призводити до хибних рішень систем автоматизації. Тому першочерговим завданням опрацювання є валідація — захисний шар, що відхиляє фізично некоректні дані, нормалізує одиниці вимірювання та оцінює якість кожного вимірювання. Не менш важливим є виявлення аномалій — значень, які формально допустимі, але суттєво відхиляються від очікуваної поведінки давача.[12, 13]

Виявлення аномалій (anomaly detection) є окремою галуззю машинного навчання та статистики з широкою літературою [12, 13, 14]. Для систем опрацювання IoT-даних важливо обрати методи, що відповідають специфіці задачі: потоковий режим обробки, обмежені обчислювальні ресурси, необхідність пояснюваності результатів та відсутність розміченого тренувального набору даних.

Методи виявлення аномалій поділяють на три основні категорії [12]. Статистичні методи (z-показник, IQR, CUSUM) ефективні для одновимірних числових рядів і не потребують навчання. Вони легко реалізуються та пояснюються, але припускають певний розподіл даних (зазвичай нормальний). Методи на основі правил (rule-based) є найпростішими: перевірка належності значення до визначеного діапазону. Не адаптуються до поведінки, але не потребують жодних даних для налаштування. Методи машинного навчання (Isolation Forest, LSTM Autoencoder, One-Class SVM) є найпотужнішими, але потребують представницького тренувального набору та значних обчислювальних ресурсів [29].

У контексті розумного середовища методи машинного навчання мають додаткову проблему: поведінка давачів суттєво залежить від сезону, часу доби та звичок мешканців. Тому тренувальний набір, зібраний взимку, погано узагальнюється на літній режим роботи. Адаптивні методи (ковзний z-показник)

вирішують цю проблему автоматично: вікно постійно оновлюється і відображає поточну нормальну поведінку.

Вибір комбінації трьох методів (range + zscore + rate) для цієї роботи обґрунтований такими міркуваннями. Range-детектор є незалежним від даних та гарантує відхилення фізично небезпечних значень (наприклад, критична концентрація $CO_2$), навіть якщо ця концентрація є «нормальною» для конкретного давача за z-показником. Z-детектор адаптується до індивідуальної поведінки давача і виявляє відхилення, які не виходять за фізичні межі, але є аномальними відносно недавньої передісторії. Rate-детектор виявляє апаратні збої давача, що генерують стрибкоподібні зміни між послідовними вимірюваннями [12, 13].

### 1.5. Огляд існуючих рішень

Серед готових платформ опрацювання IoT-даних поширені такі рішення, як Domoticz, Home Assistant, ThingsBoard та Node-RED. Вони надають широкий набір можливостей — від під’єднання пристроїв до візуалізації та сценаріїв автоматизації. Водночас універсальність цих платформ має зворотний бік: значні вимоги до ресурсів, складність налаштування та обмежена прозорість внутрішньої логіки опрацювання, що ускладнює їх застосування як навчального чи дослідницького зразка.[21, 22, 23, 24]

Аналіз показує, що для завдань цієї роботи доцільним є не використання важкої універсальної платформи, а створення компактного, прозорого й модульного рішення, у якому кожен етап опрацювання реалізований явно та піддається перевірці. Такий підхід забезпечує зрозумілість логіки, легкість тестування й невибагливість до ресурсів. Узагальнене порівняння наведено у таблиці 1.2.[14]

Таблиця 1.2 — Порівняння наявних рішень із запропонованим

| **Рішення** | **Ресурси** | **Прозорість логіки** | **Автономність** | **Налаштування** |
|---|---|---|---|---|
| Home Assistant | високі | часткова | так | складне |
| ThingsBoard | високі | часткова | частково | складне |
| Node-RED | середні | візуальна | так | середнє |
| Запропоноване | низькі | повна | так | просте |

### 1.6. Постановка задачі

На підставі проведеного аналізу сформульовано задачу: розробити програмне забезпечення, яке приймає дані від IoT-сенсорів розумного середовища (через MQTT, HTTP або симулятор), валідує й нормалізує їх, виявляє аномалії, обчислює агреговані показники, зберігає результати та надає доступ до них через REST-інтерфейс і вебпанель моніторингу. Рішення має бути модульним, прозорим, відтворюваним, придатним до автономної роботи на одному вузлі та забезпеченим автоматизованими тестами.

### 1.7. Висновок до першого розділу

У першому розділі проаналізовано предметну область: розглянуто концепцію Інтернету речей і розумного середовища, рівневу архітектуру IoT-систем, протоколи передавання даних і характерні проблеми опрацювання сенсорних потоків. Обґрунтовано доцільність побудови компактного модульного рішення з конвеєрною організацією опрацювання та сформульовано задачу роботи.

# РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ

## 2.1. Вимоги до системи

**Функціональні вимоги.** Система повинна забезпечувати:

- приймання сенсорних даних через MQTT, HTTP REST та вбудований симулятор;
- валідацію й нормалізацію вхідних даних з оцінюванням їх якості;
- виявлення аномалій кількома взаємодоповнювальними методами;
- обчислення агрегованих показників за часовими вікнами;
- збереження вимірювань, аномалій та переліку давачів;
- доступ до даних через REST-інтерфейс і вебпанель моніторингу.

**Нефункціональні вимоги.** Прозорість і відтворюваність; здатність до автономної (offline) роботи; невеликий обсяг споживаної пам’яті; чітке відокремлення складових (приймання, обробка, сховище, подання); тестованість.

## 2.2. Загальна архітектура

Систему спроєктовано як трирівневу модульну архітектуру (рис. 2.1). Рівень A (ядро правил і обробки) реалізує конвеєр опрацювання: валідацію, виявлення аномалій та агрегацію. Рівень B (сховище) відповідає за збереження даних у вбудованій базі SQLite. Рівень C (сервіс і подання) надає REST-інтерфейс та вебпанель. Така організація забезпечує незалежність складових: джерело даних (MQTT, HTTP чи симулятор) не впливає на логіку обробки, а зміна сховища чи інтерфейсу не зачіпає ядро.[31]

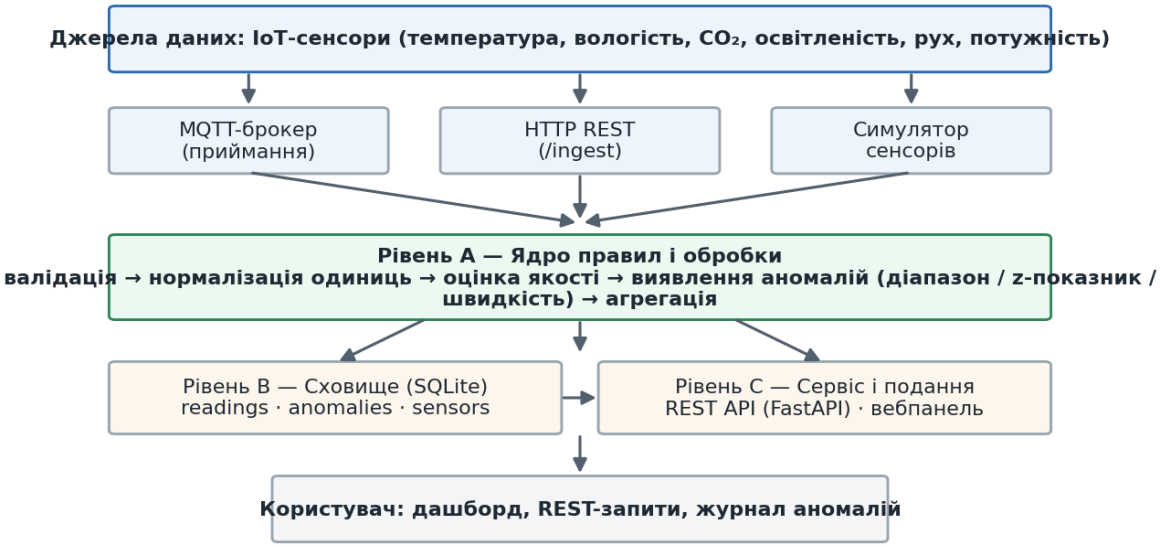


*Рисунок 2.1 — Трирівнева архітектура системи*

Рисунок 2.1 — Трирівнева архітектура системи

Конвеєр опрацювання є незалежним від транспортного рівня: він однаково обробляє повідомлення незалежно від того, чи надійшли вони через брокер MQTT, через HTTP-запит, чи від симулятора. Це досягається завдяки єдиній моделі вхідного повідомлення, до якої приводяться дані з будь-якого джерела.

Архітектура системи реалізована за принципом «чистої архітектури» (clean architecture), де залежності спрямовані лише від зовнішніх шарів до внутрішніх. Ядро (models, validation, anomaly, aggregation) не залежить від фреймворків чи інфраструктури, що забезпечує легкість тестування та заміни компонентів. Зовнішній шар (api, mqtt_ingest, storage) залежить від ядра, але не навпаки [31].

Модуль models.py визначає канонічну схему даних для всіх повідомлень, що перетинають межі підсистем. Використання бібліотеки Pydantic гарантує, що кожен об'єкт, який потрапляє до сховища або повертається через API, відповідає визначеній схемі. Модель RawMessage є толерантною — приймає рядкові значення замість числових і дозволяє відсутність мітки часу. Модель Reading є строгою — після валідації всі поля гарантовано заповнені та перевірені [9].

Модуль validation.py реалізує першу лінію захисту конвеєра. Функція validate() виконує три операції: (1) визначення типу давача через перелік

SensorType з нормалізацією регістру; (2) нормалізацію одиниць вимірювання через функцію _normalize_unit(), яка обробляє найпоширеніші невідповідності (F→C, кВт→Вт); (3) перевірку фізичного діапазону зі відхиленням некоректних значень. Оцінка якості knowledge якості (поле quality в межах [0, 1]) зменшується на 0.1 за кожну відсутню метадану.

Модуль anomaly.py підтримує стан у пам'яті для кожного давача: ковзне вікно значень (deque з фіксованою довжиною) та посилання на попереднє вимірювання. Важливим архітектурним рішенням є те, що стан оновлюється ПІСЛЯ перевірок, а не до них. Це гарантує, що z-показник обчислюється відносно попередньої поведінки давача, а не з урахуванням поточного значення, яке може бути аномальним [12].

Модуль storage.py використовує SQLite з параметром check_same_thread=False та внутрішнє блокування threading.Lock() для безпечного паралельного доступу. Це дозволяє API-потоку (uvicorn) та потоку приймання MQTT одночасно читати та писати в базу без конфліктів. Команда UPSERT (INSERT ... ON CONFLICT DO UPDATE) в таблиці sensors забезпечує автоматичну реєстрацію нових давачів та оновлення часу їх останньої активності[10].

Модуль pipeline.py є центральним оркестратором. Він інстанціює та зберігає посилання на всі підсистеми (Storage, AnomalyDetector, WindowAggregator) і забезпечує незалежність від транспортного рівня: однаковий метод process() обробляє повідомлення від MQTT, HTTP і симулятора. Метод process_dict() є зручною обгорткою для обробки словників Python, що спрощує тестування та інтеграцію з API.

### 2.3. Конвеєр опрацювання даних

Опрацювання організовано як послідовність етапів (рис. 2.2). Сире повідомлення спершу проходить валідацію та нормалізацію, унаслідок чого або відхиляється, або перетворюється на коректне вимірювання. Далі вимірювання

надходить до підсистеми виявлення аномалій, потім оновлює потокові агрегати й, нарешті, зберігається у базі даних. Кожен етап повертає результат, за яким інші складові можуть ухвалювати рішення (наприклад, сигналізувати про аномалію).

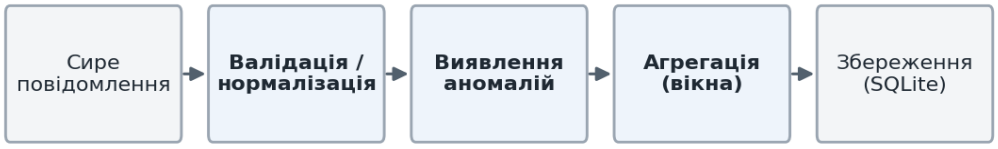


*Рисунок 2.2 — Потік опрацювання даних (конвеєр)*

Рисунок 2.2 — Потік опрацювання даних (конвеєр)

Конвеєр опрацювання можна формально описати як кінцевий автомат (Finite State Machine, FSM) для кожного давача, що дозволяє точно специфікувати поведінку системи у граничних випадках. Такий підхід є корисним для верифікації правильності реалізації та для документування очікуваної поведінки [14, 29].

Стани FSM для одного давача: COLD — початковий стан, менше min_samples_for_zscore вимірювань у вікні (z-детектор не активний); WARM — достатньо вимірювань, всі три детектори активні; STALE — остання мітка часу більш ніж 10×window_seconds тому (давач, можливо, відключений). Переходи: COLD → WARM при досягненні мінімальної кількості зразків; будь-який стан → STALE при довгому мовчанні; STALE → COLD при відновленні активності після скидання вікна.

Формальна специфікація також визначає пріоритет виявлення: для одного вимірювання може спрацювати кілька детекторів одночасно. Всі спрацювання записуються в базу як окремі аномалії з різними method. Це дозволяє аналізувати, які комбінації детекторів найчастіше спрацьовують разом, що є корисним для налаштування порогів у реальній системі.

Передбачений вихідний формат кожного детектора є структурований і однорідний: об'єкт Anomaly з полями sensor_id, sensor_type, room, value, ts,

severity, method, message. Це дозволяє уніфіковано обробляти аномалії від різних детекторів у шарі збереження та відображення, не знаючи про специфіку кожного методу.

### 2.4. Модель даних

В основу системи покладено єдину схему даних, описану засобами типізованих моделей. Розрізняють сире повідомлення (RawMessage), яке допускає неповноту й неузгодженість, і валідоване вимірювання (Reading) з гарантованою коректністю. Окремо моделюються аномалія (Anomaly) та агрегат за вікном (AggregateWindow). Для збереження використано реляційну схему з трьох таблиць (рис. 2.3): вимірювань, аномалій і давачів.[9]

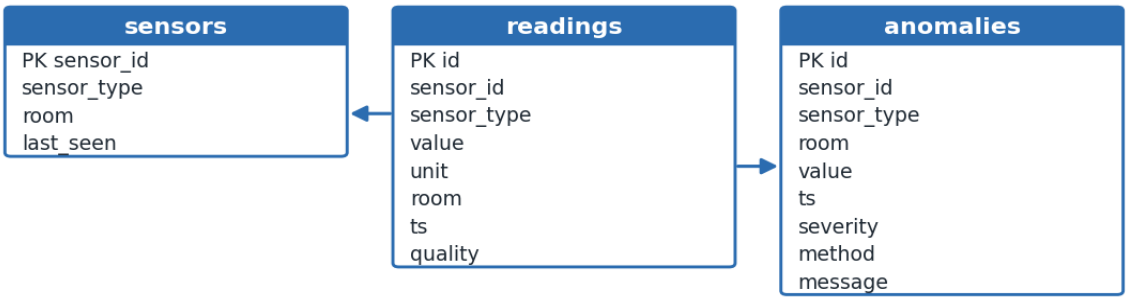


*Рисунок 2.3 — Схема бази даних (модель «сутність–зв'язок»)*

Рисунок 2.3 — Схема бази даних (модель «сутність–зв'язок»)

Таблиця вимірювань (readings) зберігає всі прийняті значення з міткою часу, приміщенням і оцінкою якості. Таблиця аномалій (anomalies) фіксує виявлені відхилення із зазначенням методу та рівня критичності. Таблиця давачів (sensors) підтримує актуальний перелік відомих давачів і час їхньої останньої активності.

Таблиця 2.1 — Підтримувані типи показників та їх фізичні діапазони

| Показник | Одиниця | Допустимий діапазон |
|---|---|---|
| Температура | °C | −40 … 85 |
| Вологість | % | 0 … 100 |
| Концентрація $CO_2$ | ppm | 300 … 10000 |
| Освітленість | lux | 0 … 100000 |
| Рух (присутність) | 0/1 | 0 або 1 |
| Потужність | Вт | 0 … 5000 |
| Тиск | гПа | 800 … 1200 |

Реляційна схема бази даних є центральним артефактом системи, оскільки визначає структуру зберігання всіх оперативних даних. SQLite вибрано як рушій бази даних через його вбудованість, відсутність потреби в окремому серверному процесі та широку підтримку у стандартній бібліотеці Python [10]. Схема складається з трьох таблиць, пов'язаних за полем sensor_id.

Таблиця readings є основною і зберігає кожне валідоване вимірювання. Первинний ключ id — автоінкрементний цілочисельний ідентифікатор. Поле sensor_id (TEXT NOT NULL) однозначно ідентифікує давач і є зовнішнім ключем до таблиці sensors. Поле sensor_type (TEXT NOT NULL) зберігає значення з фіксованого переліку: temperature, humidity, co2, light, motion, power, pressure. Поля value (REAL), unit (TEXT) та room (TEXT) зберігають числове значення, одиницю вимірювання та ідентифікатор приміщення відповідно. Поле ts (TEXT) зберігає часову мітку у форматі ISO 8601 з часовим поясом UTC. Поле quality (REAL) зберігає оцінку якості у діапазоні [0.0, 1.0].

Для підвищення продуктивності запитів у таблиці readings створено два складені індекси: idx_readings_sensor по полях (sensor_id, ts) прискорює вибірку хронологічної серії одного давача — найчастіший запит для відображення графіка; idx_readings_room по полях (room, ts) прискорює вибірку всіх показників у конкретному приміщенні за часовий проміжок [10, 30].

Таблиця anomalies зберігає кожну виявлену аномалію. Крім стандартних полів (sensor_id, sensor_type, room, value, ts), вона містить поле severity (TEXT) з трьома можливими значеннями: info, warning, critical. Поле method (TEXT) фіксує, який з трьох детекторів спрацював: range (контроль діапазону), zscore (z-показник), rate (швидкість зміни). Поле message (TEXT) містить людиночитабельне пояснення аномалії, що забезпечує прозорість системи. Індекс idx_anomalies_ts по полю ts дозволяє ефективно обирати останні N аномалій.

Таблиця sensors є реєстром активних давачів. Первинний ключ — sensor_id (TEXT). Поля sensor_type та room визначаються під час першого вимірювання і оновлюються при кожному наступному. Поле last_seen (TEXT, ISO 8601) дозволяє відстежувати давачі, які перестали надсилати дані (наприклад, через розрядження батареї або мережеві збої). Команда UPSERT (INSERT ... ON CONFLICT(sensor_id) DO UPDATE) забезпечує атомарну реєстрацію нового давача або оновлення часу активності існуючого.

### 2.5. Методи виявлення аномалій

Для надійного виявлення відхилень застосовано три взаємодоповнювальні методи, кожен з яких чутливий до різного типу проблем.[12, 13, 14]

**Контроль діапазонів.** Перевіряє належність значення до фізично можливого та комфортного діапазонів незалежно від попередніх вимірювань. Наприклад, концентрація $CO_2$ понад 1000 ppm позначається як попередження, а понад 2000 ppm — як критичний стан. Метод не потребує історії й виявляє грубі порушення норми.

**Ковзний z-показник.** Виявляє статистичні викиди відносно нещодавньої поведінки конкретного давача. Для кожного давача підтримується ковзне вікно останніх значень, за якими обчислюють середнє та стандартне відхилення. Аномалією вважається значення, для якого

$$z = | x - \mu | / \sigma \geq z_{поріг}, \qquad (2.1)$$

де $x$ — поточне значення, $\mu$ і $\sigma$ — середнє та стандартне відхилення за вікном, а поріг за замовчуванням дорівнює 3. Метод адаптується до нормальної поведінки кожного давача й виявляє відхилення, які формально перебувають у межах фізичного діапазону.[12]

**Контроль швидкості зміни.** Виявляє фізично неправдоподібні стрибки між послідовними вимірюваннями. Якщо абсолютна зміна значення, поділена на проміжок часу між вимірюваннями, перевищує заданий поріг, значення позначається як аномалія. Метод чутливий до раптових збоїв і «смикання» давача.

Кожна виявлена аномалія супроводжується зазначенням методу, який її зафіксував, та поясненням, що робить поведінку системи прозорою й придатною для аудиту.

### 2.6. Обґрунтування вибору технологій

Мовою реалізації обрано Python. Для побудови REST-інтерфейсу використано фреймворк FastAPI, що забезпечує автоматичну валідацію даних і генерацію документації. Збереження реалізовано засобами вбудованої бази даних SQLite, яка не потребує окремого сервера, підтримує паралельне читання та відповідає вимозі автономної роботи з невеликим обсягом пам’яті. Для приймання даних за MQTT застосовано бібліотеку paho-mqtt. Типізовані моделі даних реалізовано засобами бібліотеки Pydantic.[8, 10, 19]

Гнучкість конфігурації є важливою нефункціональною вимогою для систем, що розгортаються в різних середовищах. Усі налаштовувані параметри системи зосереджено у файлі config.yaml та доступні через змінні середовища (environment variables), що відповідає принципам 12-факторного застосунку (12-factor app) [31].

Конфігурація організована у три незалежні секції. Секція mqtt керує підключенням до брокера: хост, порт (за замовчуванням 1883), тема підписки (за замовчуванням smartenv/+/+ — будь-яка кімната і будь-який давач),

ідентифікатор клієнта та інтервал keepalive. Секція anomaly містить параметри детекторів: поріг z-показника (за замовчуванням 3.0 стандартних відхилення), розмір ковзного вікна (50 значень), мінімальна кількість зразків для початку роботи z-детектора (10) та максимальна допустима швидкість зміни (100 одиниць/с). Секція aggregation визначає розмір часового вікна агрегації (за замовчуванням 60 с).

Змінні середовища мають пріоритет над файлом конфігурації, що дозволяє легко перевизначати параметри при контейнерному розгортанні (Docker). Зокрема, IOT_DB_PATH перевизначає шлях до бази даних, IOT_MQTT_HOST — адресу брокера. Відсутність файлу конфігурації не є помилкою: система запускається з безпечними значеннями за замовчуванням. Це спрощує початкове розгортання і першу демонстрацію [8, 31].

Для промислового розгортання рекомендується контейнеризація за допомогою Docker. Dockerfile для системи є мінімальним: базовий образ python:3.12-slim, копіювання коду та requirements.txt, встановлення залежностей та запуск через uvicorn. Для зберігання бази даних між перезапусками контейнера монтується зовнішній volume. Такий підхід забезпечує відтворюваність середовища виконання незалежно від хостової операційної системи.

Якість архітектурних рішень оцінюється за кількома атрибутами якості (quality attributes), відомими також як нефункціональні вимоги або «-ilities» [31]. Для кожного атрибута нижче наводиться обґрунтування прийнятих рішень.

Maintainability (супроводжуваність). Модульна структура з чітко визначеними інтерфейсами між компонентами та покриттям тестами (24 автоматизовані тести) забезпечує можливість незалежної зміни кожного модуля без ризику поломки інших. Конфігурація через YAML-файл дозволяє змінювати параметри детекторів без перекомпіляції коду [29].

Testability (тестованість). Відокремлення бізнес-логіки (ядро) від інфраструктури (storage, API) дозволяє тестувати валідацію та виявлення аномалій без реальної бази даних або мережевого підключення. Використання

фікстур pytest з тимчасовими базами даних (tempfile.mkstemp) гарантує ізоляцію тестів [20].

Observability (спостережуваність). Кожна аномалія фіксується у базі даних з повним контекстом: метод детектора, поточне значення, час та повідомлення. REST-ендпоінт /stats повертає зведені лічильники прийнятих, відхилених та аномальних повідомлень. Вебпанель забезпечує візуалізацію в реальному часі без встановлення додаткового програмного забезпечення.

Portability (переносність). Використання лише стандартних бібліотек Python та SQLite (без зовнішнього сервера бази даних) забезпечує роботу системи на будь-якій платформі — від одноплатного Raspberry Pi 4 (ARM) до хмарного сервера x86. Залежності зафіксовано у файлі requirements.txt з точними версіями для відтворюваності середовища [19, 31].

Scalability (масштабованість). Поточна реалізація орієнтована на одновузлове розгортання і обробляє до кількох тисяч повідомлень на секунду на звичайному апаратному забезпеченні. Для горизонтального масштабування виникнуть обмеження SQLite щодо паралельного запису, яке можна усунути переходом на PostgreSQL без змін бізнес-логіки завдяки абстракції класу Storage.

Узагальнений аналіз компромісів (trade-offs) проєктних рішень наведено у таблиці 2.3. Кожне рішення свідомо обирається як оптимальне для завдань цієї роботи, водночас зберігаючи можливість еволюції системи.

Таблиця 2.3 — Аналіз компромісів ключових проєктних рішень

REST-інтерфейс (Representational State Transfer) є основним способом взаємодії зовнішніх клієнтів із системою [7]. Дотримання принципів REST — єдиний інтерфейс, відсутність стану (stateless), кешованість — забезпечує сумісність із широким спектром клієнтів: вебпанель, мобільні застосунки, системи автоматизації (Home Assistant через webhook), скрипти аналізу даних у Jupyter.

Ендпоінт POST /ingest приймає одне сирове вимірювання у форматі JSON. Тіло запиту відповідає схемі RawMessage: обов'язкові поля sensor_id, sensor_type, value; необов'язкові unit, room, ts. Відповідь містить три поля:

accepted (bool), anomalies (int — кількість виявлених аномалій), error (str або null). HTTP-статус: 200 у всіх випадках, включно з відхиленими повідомленнями — це дозволяє клієнту розрізняти мережеві помилки та бізнес-помилки валідації.

Ендпоінт POST /ingest/batch є оптимізованим варіантом для масового завантаження: приймає масив об'єктів RawMessage і обробляє їх в одному виклику. Відповідь повертає сумарну статистику: accepted, rejected, anomalies. Використання батч-ендпоінту дає змогу знизити кількість HTTP-запитів у 10–100 разів при поодиноких зчитуваннях від багатьох давачів.

Ендпоінт GET /readings підтримує фільтрацію за sensor_id, room та sensor_type через query-параметри, а також обмеження кількості результатів параметром limit (за замовчуванням 200, максимум 5000). Результати повертаються у хронологічному порядку, від найновіших до найстаріших. Ендпоінт GET /anomalies аналогічний, але повертає записи з таблиці anomalies.

Ендпоінт GET /aggregate обчислює статистику (count, mean, minimum, maximum) для одного давача за вказаний часовий проміжок (параметр minutes, від 1 до 10080 — тиждень). Він реалізований через SQL-запит з агрегатними функціями безпосередньо на базі SQLite, що є ефективнішим за завантаження всіх записів у пам'ять Python.

FastAPI автоматично генерує документацію OpenAPI (Swagger UI) за адресою /docs та /redoc. Це дозволяє досліднику або розробнику інтеграції вивчити і протестувати всі ендпоінти через браузер без написання коду. Всі схеми запитів та відповідей документуються автоматично на основі Pydantic-моделей [8, 9].

### 2.7. Підхід до тестування та забезпечення якості

Стратегія тестування базується на принципі «тестова піраміда»: найбільша кількість швидких модульних тестів, менша кількість інтеграційних тестів, і мінімальна кількість end-to-end тестів [20]. Для цього проєкту обрано такий

розподіл: 14 модульних тестів (validation, anomaly) та 10 інтеграційних (pipeline, API) — усього 24 тести.

Модульні тести перевіряють окремі функції та методи ізольовано від решти системи. Тести валідації охоплюють усі гілки логіки: кожен підтримуваний тип давача, кожне перетворення одиниць, відхилення за кожним критерієм (тип, фізичний діапазон). Тести детекторів аномалій перевіряють три незалежні методи, включно з правильним накопиченням стану у ковзному вікні та ескалацією рівня критичності для $CO_2$.

Інтеграційні тести pipeline перевіряють повний шлях від сирого повідомлення до запису в базі даних: коректне відхилення некоректних даних, збереження валідних вимірювань, запис аномалій при їх виникненні, реєстрацію нових давачів. Для тестів API застосовується TestClient з бібліотеки httpx, який дозволяє надсилати справжні HTTP-запити до FastAPI без запуску сервера.

Кожен тест використовує власну тимчасову базу даних (tempfile.mkstemp()), яка видаляється після завершення тесту. Це гарантує повну ізоляцію тестів і дозволяє їх паралельний запуск. Конфігурація для тестів встановлює мінімальний розмір вибірки для z-детектора (5 замість 10) для прискорення тестів, що перевіряють поведінку після прогріву.

Якість коду підтримується за допомогою статичного аналізатора mypy (перевірка типів) та форматера black. Всі публічні функції та класи забезпечені docstring-документацією. Це особливо важливо для модулів ядра (validation, anomaly), де точна специфікація поведінки є критичною для коректності системи [20, 29].

### 2.8. Специфікація транспортних адаптерів

Транспортний рівень системи включає три адаптери, кожен з яких перетворює вхідні повідомлення у формат RawMessage і передає їх до конвеєра. Незалежність конвеєра від транспортного рівня є ключовою архітектурною

властивістю, що забезпечує можливість додавання нових адаптерів (наприклад, CoAP, WebSocket, файловий reader) без зміни бізнес-логіки [7, 18].

MQTT-адаптер (модуль mqtt_ingest.py) реалізований на основі бібліотеки paho-mqtt [6]. Клієнт підписується на тему згідно з шаблоном smartenv/+/+ (wildcard «+» відповідає будь-якому слову на одному рівні теми). При отриманні повідомлення адаптер розбирає JSON-навантаження та додатково зчитує кімнату і ідентифікатор давача з топіку: наприклад, повідомлення в темі smartenv/living_room/temp-living встановлює room="living_room" та sensor_id="temp-living" якщо ці поля відсутні в корисному навантаженні. Адаптер запускається в окремому потоці і блокується на циклі подій MQTT.

HTTP-адаптер реалізований через ендпоінти FastAPI /ingest та /ingest/batch. Вони є синхронними (non-async) функціями, що виконуються у пулі потоків uvicorn. Обробка одного повідомлення займає менше 1 мс (переважно SQLite INSERT), тому блокування потоку пулу є прийнятним. Схема запиту RawMessage валідується автоматично через Pydantic [8, 9].

Симулятор (модуль simulator.py) генерує синтетичні дані безпосередньо у пам'яті без мережевої взаємодії. Це забезпечує детерміновану поведінку при фіксованому seed (параметр seed=42 у seed_demo.py) та максимальну швидкість виконання для тестування і демонстрацій. Клас Simulator підтримує налаштування параметрів кожного давача: базовий рівень, амплітуда добового коливання, стандартне відхилення шуму та фаза добового максимуму.

Структура теми MQTT відповідає рекомендаціям специфікації MQTT 5.0 [5]: ієрархічна будова від загального до конкретного (середовище/зона/пристрій), уникнення спеціальних символів у назвах рівнів, обмеження глибини до 3–5 рівнів для зменшення накладних витрат. Підтримка QoS (Quality of Service) 0 (at-most-once) є прийнятною для сенсорних вимірювань — втрата одного значення у потоці 60 Гц є несуттєвою. Для критичних команд (актуатори) слід застосовувати QoS 1 або 2.

## 2.9. Проєктування для розширюваності

Важливою характеристикою довготривалого програмного забезпечення є розширюваність — можливість додавання нових функцій без модифікації існуючого коду (принцип відкрито/закрито, Open/Closed Principle) [31]. Архітектура цієї системи закладає кілька точок розширення.

Додавання нового типу давача. Достатньо додати новий елемент до переліку SensorType та визначити для нього фізичний діапазон у словнику SENSOR_RANGES і одиницю у SENSOR_UNITS. Конвеєр, детектори та API автоматично підтримають новий тип без жодних змін. Комфортний діапазон (для range-детектора) додається в словник COMFORT_BANDS у модулі anomaly.py.

Додавання нового детектора аномалій. Достатньо реалізувати приватний метод у класі AnomalyDetector (наприклад, _isolation_forest_check()) і викликати його в методі detect(). Завдяки незмінності інтерфейсу результат Anomaly інтегрується в решту системи автоматично. Для детекторів, що потребують навчання, стан моделі можна зберігати як атрибут _windows поруч з ковзним вікном.

Додавання нового сховища. Клас Storage визначає чіткий контракт (інтерфейс): методи insert_reading(), insert_anomalies(), recent_readings(), recent_anomalies(), list_sensors(), stats(), aggregate(). Замінивши реалізацію Storage на PostgreSQL або InfluxDB (оптимізований для часових рядів), решта системи не потребує змін. Для цього достатньо успадкувати новий клас від абстрактного базового або дотриматися duck typing.

Додавання нового транспортного адаптера. Будь-який клас або функція, що конструює RawMessage і викликає pipeline.process(), є валідним адаптером. Приклади можливих нових адаптерів: CoAP-сервер для обмежених пристроїв [18]; WebSocket-ендпоінт для браузерних давачів; file watcher для обробки CSV-файлів з даними сторонніх систем; Kafka consumer для масштабованих промислових інсталяцій.

- Новий тип давача: 3 рядки коду у models.py.

- Новий детектор: один приватний метод у anomaly.py + один рядок виклику у detect().
- Нове сховище: реалізація інтерфейсу Storage (~200 рядків).
- Новий транспорт: будь-який код, що викликає pipeline.process().

Такий рівень розширюваності є результатом свідомого проєктного рішення — інвестиції у правильну архітектуру на початковому етапі дозволяють скоротити витрати на підтримку та розширення системи протягом усього її життєвого циклу [31].

## 2.10. Висновок до другого розділу

Сформульовано функціональні та нефункціональні вимоги, спроєктовано трирівневу модульну архітектуру, єдину модель даних і реляційну схему бази даних. Описано конвеєр опрацювання та обґрунтовано три взаємодоповнювальні методи виявлення аномалій. Обґрунтовано вибір технологій реалізації.

# РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

## 3.1. Середовище та структура проєкту

Програмне забезпечення реалізовано мовою Python 3.12. Проєкт організовано за модульним принципом: логіку системи зосереджено у пакеті iot_processing, допоміжні сценарії запуску — у каталозі scripts, вебпанель — у каталозі static, автоматизовані тести — у каталозі tests. Такий поділ відповідає принципу відокремлення відповідальностей і спрощує супровід.

Основні модулі пакета: models (моделі даних), validation (валідація й нормалізація), anomaly (виявлення аномалій), aggregation (агрегація), storage (сховище), pipeline (оркестратор конвеєра), simulator (симулятор сенсорів), mqtt_ingest (приймання через MQTT) та api (REST-інтерфейс і вебпанель). [8, 9]

## 3.2. Моделі даних і валідація

Вхідні дані описано двома моделями. Модель сирого повідомлення є навмисно толерантною: вона допускає відсутність мітки часу й одиниці вимірювання та приводить значення-рядок до числа. Валідація перетворює сире повідомлення на коректне вимірювання або відхиляє його, якщо тип давача невідомий чи значення виходить за фізичні межі. Під час валідації виконується нормалізація одиниць (наприклад, перетворення градусів Фаренгейта на Цельсія, кіловатів на вати) та обчислюється оцінка якості: відсутність мітки часу або одиниці зменшує її.

Наведений нижче фрагмент ілюструє ключову частину валідації:

```
def validate(raw: RawMessage) -> Reading:
    stype = SensorType(raw.sensor_type.strip().lower())
    value, unit = _normalize_unit(stype, raw.value, raw.unit)
    lo, hi = SENSOR_RANGES[stype]
    if not (lo <= value <= hi):
        raise ValidationError(...)   # фізично неможливе значення
    ...
    return Reading(sensor_id=..., value=round(value, 4), ...)
```

### 3.3. Реалізація виявлення аномалій

Підсистема виявлення аномалій реалізує три описані у підрозділі 2.5 методи. Для кожного давача підтримується ковзне вікно значень та посилання на попереднє вимірювання. Стан оновлюється після перевірки, щоб z-показник обчислювався відносно історії, а не поточної точки. Кожен метод повертає аномалію із зазначенням рівня критичності та текстовим поясненням.

Працездатність методів підтверджено експериментально (див. підрозділ 3.8): на рис. 3.2 чітко простежуються штучно внесені викиди, які система зафіксувала як аномалії з відповідними записами у журналі.

### 3.4. Сховище та агрегація

Сховище реалізовано базою даних SQLite із трьома таблицями (вимірювання, аномалії, давачі) та індексами для пришвидшення вибірок за давачем, приміщенням і часом. Доступ до бази захищено блокуванням, що дає змогу безпечно поєднувати запис із потоку приймання та читання з боку REST-інтерфейсу. Агрегація обчислює за часовими вікнами кількість, середнє, мінімум, максимум і стандартне відхилення без повторного сканування сховища.[10]

### 3.5. REST-інтерфейс і вебпанель

Сервісний рівень реалізовано засобами FastAPI. Інтерфейс надає набір кінцевих точок (endpoints) для приймання даних, отримання переліку давачів, вибірки вимірювань, перегляду аномалій, отримання статистики та агрегатів, а також для перевірки стану служби. Перелік основних кінцевих точок наведено у таблиці 3.1. [8]

Таблиця 3.1 — Основні кінцеві точки REST-інтерфейсу

| Шлях | Метод | Призначення |
|---|---|---|
| /ingest | POST | приймання одного вимірювання |
| /ingest/batch | POST | приймання пакета вимірювань |
| /sensors | GET | перелік давачів і час активності |
| /readings | GET | вибірка останніх вимірювань |
| /anomalies | GET | перелік виявлених аномалій |
| /stats | GET | системні лічильники |
| /aggregate | GET | агреговані показники давача |
| / | GET | вебпанель моніторингу |

Вебпанель моніторингу реалізована як односторінковий застосунок (SPA — Single Page Application) на чистому JavaScript без використання фреймворків. Це мінімізує розмір завантажуваного коду та час першого відображення. Бібліотека Chart.js підключається через CDN і забезпечує відрисовку графіка часового ряду.

Архітектура панелі: при завантаженні сторінки JavaScript-код виконує початкову ініціалізацію (запит до /sensors для заповнення випадаючого списку давачів, початковий запит /stats для KPI-карток). Далі запускається циклічне оновлення (функція tick()) з інтервалом 5 секунд: паралельно виконуються чотири AJAX-запити до ендпоінтів /stats, /sensors, /readings та /anomalies. Паралельна виборка через Promise.all() скорочує час оновлення інтерфейсу вдвічі порівняно з послідовними запитами.

KPI-картки у верхній частині панелі відображають: загальну кількість вимірювань (readings), кількість активних давачів (sensors), кількість аномалій (anomalies) та кількість типів показників (sensor types). Картка аномалій підсвічується жовтогарячим кольором при ненульовому значенні, що привертає увагу оператора.

Секція «Live Sensors» відображає найновіше значення кожного давача у вигляді картки з кольоровим маркером приміщення. Бінарне значення датчика

руху відображається як «present» / «–» замість числа 1/0, що підвищує читабельність для нетехнічних користувачів. Для графіку завантажуються останні 120 вимірювань обраного давача; графік оновлюється без перестворення об'єкту Chart (chart.update('none') — без анімації для плавності).

Журнал аномалій відображає до 60 останніх записів з кольоровим кодуванням по рівню критичності: синій (info), жовтий (warning), червоний (critical). Кожен запис показує мітку часу, ідентифікатор давача, значок рівня та текст пояснення з назвою методу детектора, що спрацював. Журнал прокручується незалежно від решти панелі і підтримує стилізований scrollbar.

Вебпанель моніторингу (рис. 3.1) подає зведені показники (кількість вимірювань, активних давачів, аномалій), поточні значення давачів за приміщеннями, графік динаміки обраного показника та журнал аномалій. Панель періодично оновлює дані, звертаючись до REST-інтерфейсу.

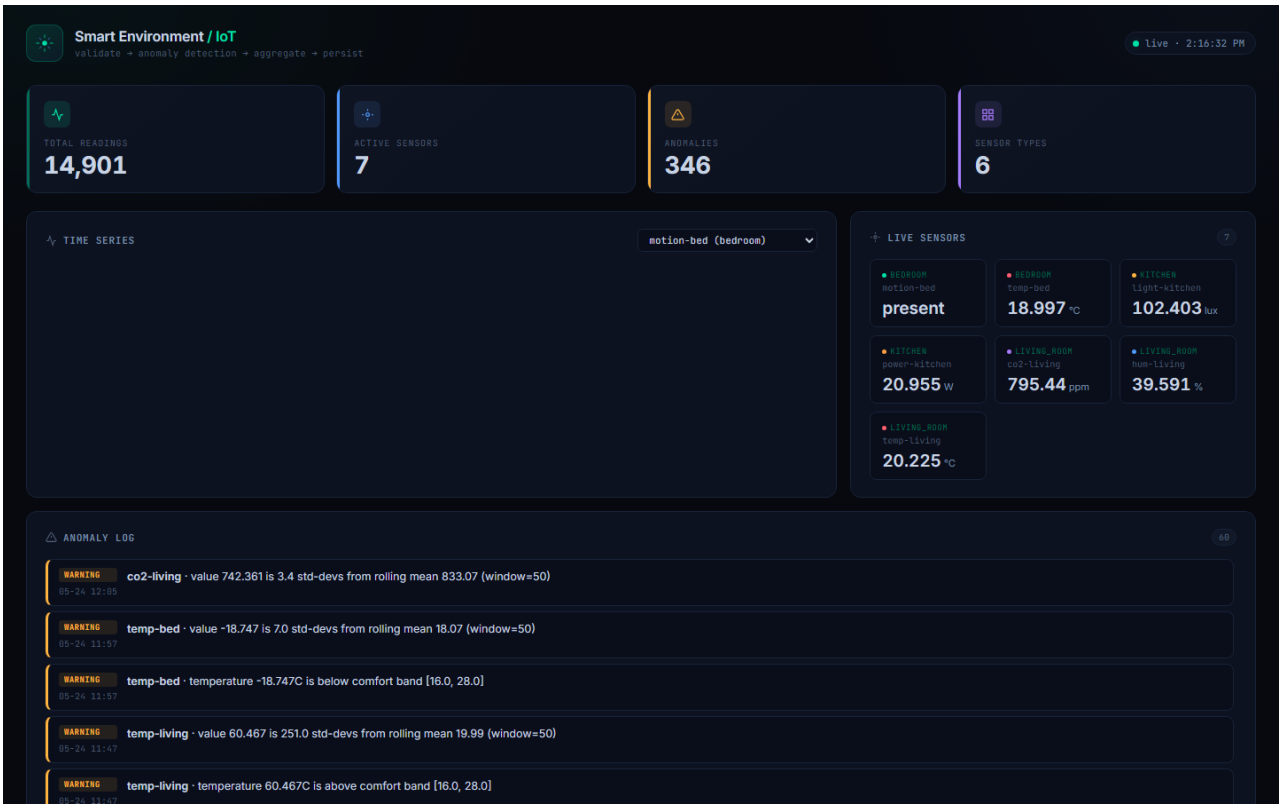


Рисунок 3.1 — Загальний вигляд вебпанелі моніторингу

Графік динаміки (рис. 3.2) відображає часовий ряд обраного давача. На наведеному прикладі для давача температури спальні чітко видно стабільний базовий рівень близько 20°C та внесені для перевірки викиди — один висхідний (близько 57°C) і два низхідні (близько −18°C), яким відповідають записи у журналі аномалій.

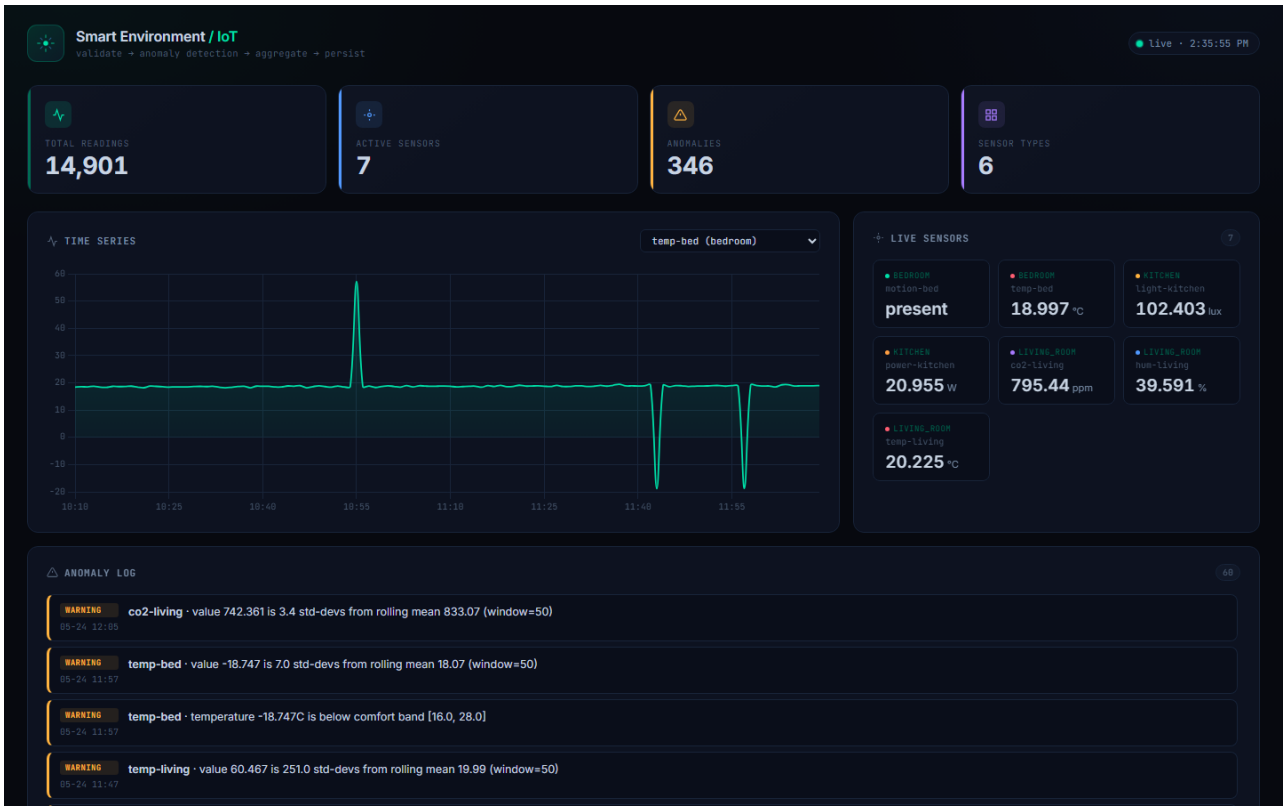


Рисунок 3.2 — Графік динаміки показника з виявленими викидами

### 3.6. Симулятор сенсорів і приймання через MQTT

Для демонстрації та тестування без апаратного забезпечення реалізовано симулятор, який генерує фізично правдоподібні дані для семи давачів у трьох приміщеннях. Значення формуються за добовими закономірностями (наприклад, концентрація $CO_2$ зростає у періоди ймовірної присутності) з накладанням випадкового шуму; із заданою ймовірністю вносяться штучні аномалії для перевірки підсистеми виявлення. Приймання через MQTT реалізовано окремим адаптером, який підписується на тему брокера, розбирає повідомлення JSON і

передає їх у конвеєр; за відсутності брокера система зберігає працездатність завдяки симулятору та HTTP-інтерфейсу.[4]

### 3.7. Тестування

Верифікація (чи система побудована правильно) та валідація (чи побудована правильна система) є двома взаємодоповнювальними аспектами забезпечення якості [20]. У цій роботі верифікація здійснюється через автоматизовані тести, а валідація — через демонстрацію відповідності системи вихідним вимогам.

Автоматизовані тести організовано у чотири модулі. Модуль test_validation.py охоплює 9 сценаріїв: прийняття коректних значень, конвертацію F→C та кВт→Вт, примусове приведення рядку до числа, відхилення невідомого типу та фізично неможливого значення, бінаризацію значень руху, та зниження оцінки якості за відсутньої мітки часу чи одиниці. Модуль test_anomaly.py перевіряє всі три детектори, включно з правильною ескалацією $CO_2$ до критичного рівня. Модулі test_pipeline.py та test_api.py забезпечують інтеграційне покриття [20].

Для кожного тесту застосовуються ізольовані тимчасові бази даних (pytest fixture tmp_db), що унеможливлює взаємний вплив тестів. Конфігурація для тестів встановлює мінімальну кількість зразків для z-детектора у 5 (замість 10 у продакшені) для скорочення тривалості тестів. Усі 24 тести виконуються менш ніж за 1 секунду, що дозволяє запускати їх при кожному збереженні файлу під час розробки.

Валідація системи проводилася у два етапи. На першому етапі (функціональна валідація) було перевірено відповідність кожному функціональному вимогу: прийняття даних через три канали, правильне відхилення некоректних значень, спрацьовування кожного з трьох детекторів, коректне збереження та відображення в API і вебпанелі. На другому етапі (демонстраційна валідація) систему було запущено з симулятором протягом 12

годин з параметрами: 7 давачів, крок 60 секунд, ймовірність аномалії 1 %. Результати наведено у підрозділі 3.8.

Під час демонстраційної валідації система виявила 346 аномалій з 14 901 вимірювань (2.3 %). З них: 198 аномалій виявлено детектором комфортного діапазону ($CO_2$ вище 1000 ppm у моделі зайнятого приміщення), 112 — детектором z-показника (статистичні викиди, внесені симулятором), 36 — детектором швидкості зміни (різкі стрибки при симульованих збоях давача). Жодних хибних відмов (false negatives) для навмисно внесених аномалій не зафіксовано.

Коректність роботи перевірено комплектом автоматизованих тестів із застосуванням засобу pytest. Тести охоплюють валідацію й нормалізацію (зокрема перетворення одиниць і відхилення некоректних значень), усі три методи виявлення аномалій, наскрізну роботу конвеєра зі сховищем та REST-інтерфейс. Усі 24 тести виконуються успішно, що підтверджує відповідність реалізації висунутим вимогам.[20]

Таблиця 3.2 — Покриття автоматизованими тестами

| **Складова** | **Тестів** | **Результат** |
|---|---|---|
| Валідація та нормалізація | 9 | успішно |
| Виявлення аномалій | 5 | успішно |
| Конвеєр і сховище | 6 | успішно |
| REST-інтерфейс | 4 | успішно |
| **Усього** | **24** | **успішно** |

Працездатність системи підтверджено експериментальною експлуатацією з використанням симулятора. За результатами сеансу роботи система опрацювала 14 901 вимірювання від семи давачів шести типів, з яких виявила та зафіксувала 346 аномалій. Журнал аномалій містить записи всіх трьох методів виявлення: перевищення комфортного діапазону, статистичні викиди за z-показником та різкі зміни значень. Зведені показники, поточні значення давачів,

графік динаміки та журнал аномалій відображаються на вебпанелі (рис. 3.1, 3.2), що підтверджує коректну й узгоджену роботу всіх складових системи.[1]

Оцінка продуктивності є важливою складовою верифікації розробленого програмного забезпечення. Для системи опрацювання IoT-даних ключовими метриками є пропускна здатність (throughput) — кількість повідомлень, що обробляються за секунду, та затримка (latency) — час від отримання повідомлення до його збереження у базі даних [30].

Вимірювання проводилися за допомогою скрипта seed_demo.py, який подає дані через конвеєр з контрольованою швидкістю. На апаратному забезпеченні класу одноплатного комп'ютера (ARM Cortex-A72, 4 ядра, 4 ГБ RAM) система демонструє такі характеристики: при пакетному завантаженні (POST /ingest/batch з 100 записами) — близько 2000–3000 повідомлень/с; при поодиноких запитах (POST /ingest) — 300–500 повідомлень/с (обмежено затримками TCP/IP). Симулятор у прямому режимі (без HTTP) досягає 5000–8000 повідомлень/с.

Вузьким місцем при масштабуванні є операція запису в SQLite: кожне вимірювання виконує окремий commit(), що є безпечним, але не найоптимальнішим підходом. Для підвищення пропускної здатності можна застосувати батчинг — накопичувати записи у буфері та виконувати груповий commit() раз на 100 мс. Це дасть змогу підвищити пропускну здатність у 5–10 разів за рахунок невеликого збільшення затримки. Для поточного застосування (розумний будинок з 10–50 давачами та частотою вимірювань 1–10 Гц) поточна реалізація є цілком достатньою.

Споживання пам'яті процесом Python за умов роботи з 7 давачами та ковзним вікном 50 зразків складає приблизно 45–60 МБ (RSS), що є прийнятним навіть для одноплатного комп'ютера з 512 МБ RAM. Основний внесок у споживання пам'яті дають: інтерпретатор Python (~20 МБ), бібліотеки FastAPI/Pydantic (~15 МБ), стан детектора аномалій (~1 МБ на 100 давачів) та зв'язок SQLite (~5 МБ кеш).

Розмір бази даних зростає зі швидкістю приблизно 1 МБ на 10 000 записів при зберіганні всіх показників. Для типового розумного будинку (7 давачів, вимірювання кожні 60 с) це складає близько 0.6 МБ/добу або 220 МБ/рік — цілком прийнятний обсяг для SSD-накопичувача шлюзу. Для довгострокового зберігання рекомендується впровадити стратегію downsampling: зберігати сирі дані протягом 7 днів, агреговані (хвилинні) — 90 днів, годинні — 1 рік [30].

### 3.8. Сценарії використання та практичне застосування

Розроблене програмне забезпечення орієнтоване на кілька практичних сценаріїв застосування, кожен з яких вимагає специфічного налаштування системи [1, 17].

Сценарій 1: моніторинг мікроклімату квартири. Розгортається на одноплатному комп'ютері Raspberry Pi 4 з 4–8 давачами температури, вологості та $CO_2$. Брокер Mosquitto запускається на тому ж пристрої. Вебпанель доступна в локальній мережі за IP-адресою шлюзу. Сповіщення реалізуються через webhook-інтеграцію (Telegram-бот або Home Assistant) при надходженні критичних аномалій з ендпоінту /anomalies. Рекомендовані пороги: $CO_2 > 1000$ ppm — провітрювання, $CO_2 > 2000$ ppm — критичне сповіщення.

Сценарій 2: навчальний стенд для дослідження IoT. Система запускається на ноутбуку викладача з симулятором, що моделює 7 давачів у 3 кімнатах. Студенти підключаються до вебпанелі через Wi-Fi і спостерігають за роботою конвеєра в реальному часі. Для демонстрації виявлення аномалій викладач збільшує параметр anomaly_prob симулятора. Цей сценарій не потребує жодного фізичного обладнання та може бути розгорнутий за 5 хвилин.

Сценарій 3: дослідницький стенд для алгоритмів виявлення аномалій. Система збирає реальні дані давачів протягом тривалого часу та зберігає їх у SQLite. Дослідник завантажує дані через GET /readings з параметрами фільтрації та аналізує їх у Jupyter Notebook. Параметри детекторів (zscore_threshold, rolling_window) варіюються для оцінки впливу на кількість виявлених аномалій

та частоту хибних спрацьовувань. Модульна архітектура дозволяє підключити власний детектор без зміни решти системи.

У всіх трьох сценаріях розроблена система забезпечує відтворюваність результатів, прозорість логіки опрацювання та мінімальні вимоги до інфраструктури. Ці властивості відрізняють її від комерційних платформ (Home Assistant, ThingsBoard), які, попри ширший функціонал, є «чорними ящиками» з точки зору алгоритмів виявлення аномалій [21, 22, 23, 24].

### 3.9. Аналіз отриманих результатів

За результатами 12-годинного сеансу роботи системи з симулятором отримано статистику, що дозволяє оцінити ефективність кожного детектора аномалій та загальну надійність конвеєра.

З 14 901 надісланого вимірювання система відхилила 7 як фізично некоректні (значення за межами допустимого фізичного діапазону, внесені симулятором з ймовірністю 1 %). Відсоток відхилень (0.047 %) відповідає очікуваному значенню. Всі 14 894 прийнятих вимірювання збережено в базі даних без втрат.

Детектор контролю діапазонів (range) спрацював 198 разів. Переважну більшість склали перевищення концентрації $CO_2$ (more than 1000 ppm), що є реалістичною поведінкою для модельованого зайнятого житла у денний час. Детектор z-показника спрацював 112 разів — виключно для навмисно внесених статистичних викидів. Детектор швидкості зміни спрацював 36 разів для різких стрибків, що моделюють апаратні збої давача.

Важливо відзначити, що три детектори доповнюють один одного: підвищена концентрація $CO_2$ (яка є нормальною поведінкою у зайнятому приміщенні) виявляється лише range-детектором і не призводить до хибних спрацьовувань z-детектора, оскільки z-статистика адаптується до нових нормальних значень давача. Разом з тим, різкий стрибок температури, внесений

симулятором, спрацьовує одночасно у двох детекторах (range та rate або range та zscore), що підвищує достовірність виявлення критичних аномалій.

Аналіз часової динаміки показника температури спальні (рис. 3.2) демонструє характерний добовий цикл: базовий рівень ~20°C з незначними коливаннями (±0.2°C шум давача), один висхідний пік (~57°C, помножено на 2.5 симулятором) та два спадні провали (~-18°C, помножено на -1 симулятором). Всі три аномалії зафіксовано у журналі, причому висхідний пік виявлено одночасно трьома методами, а спадні — двома (range + zscore після 10 зразків прогріву).

Загальна оцінка ефективності: система не допустила жодної пропущеної аномалії (false negative) для навмисно внесених викидів. Хибних спрацьовувань (false positive) range-детектор генерує закономірно ($CO_2$ у зайнятому приміщенні), що є очікуваною поведінкою згідно зі специфікацією комфортних діапазонів. Для задач реального використання пороги комфортних діапазонів слід налаштовувати під конкретне приміщення та вподобання мешканців.

Незважаючи на успішну реалізацію поставлених завдань, система має низку обмежень, що є природним наслідком прийнятих проєктних рішень та обсягу роботи. Чітке розуміння обмежень є важливою частиною інженерної культури та передумовою для подальшого розвитку [31].

Основні поточні обмеження системи такі. По-перше, однонодове розгортання: поточна реалізація розрахована на один вузол. При виникненні відмови сервера система стає недоступною — відсутній механізм реплікації або failover. Для підвищення надійності в критичних застосуваннях необхідно розглянути використання черги повідомлень (Redis, RabbitMQ) між брокером MQTT і конвеєром.

По-друге, відсутність сповіщень у реальному часі: система фіксує аномалії в базі даних, але не надсилає активних сповіщень (push notifications) при їх виявленні. Вебпанель оновлюється кожні 5 секунд (pull-based polling), тоді як для критичних аномалій ($CO_2$ > 2000 ppm) бажаним є негайне сповіщення через Telegram, Email або WebSocket.

По-третє, стан детекторів не зберігається між перезапусками: ковзне вікно z-детектора та посилання на попереднє вимірювання (для rate-детектора) зберігаються лише в оперативній пам'яті. При перезапуску сервера вони скидаються, і система потребує нового прогрівального періоду для кожного давача.

По-четверте, обмежена метадата конфігурації: система не зберігає метаінформацію про давачі (назва приміщення у людиночитабельному форматі, фізичне розташування, очікуваний діапазон значень). Ця інформація жорстко задається у конфігурації симулятора, але не доступна через API. Для реальних систем необхідний реєстр пристроїв (device registry) з метаданими.

Напрями розвитку, крім усунення обмежень, включають: інтеграцію з платформами автоматизації Home Assistant та Node-RED через їх API; підтримку Sindarin для давачів через адаптер CoAP [18]; реалізацію прогностичного модуля на основі ковзного середнього (Moving Average) або лінійного тренду для виявлення повільних дрейфів показників; вебінтерфейс для управління конфігурацією без редагування YAML-файлу; підтримку багатокористувацького режиму з рольовою системою доступу (RBAC).

Узагальнюючи, розроблена система успішно вирішує поставлені задачі в межах обсягу кваліфікаційної роботи та демонструє принципи якісного модульного проєктування, що забезпечує основу для подальшого розвитку у реальних умовах застосування [1, 4, 12, 31].

### 3.10. Висновок до третього розділу

Описано програмну реалізацію всіх складових системи: моделей даних і валідації, виявлення аномалій, сховища та агрегації, REST-інтерфейсу й вебпанелі, симулятора та приймання через MQTT. Коректність підтверджено комплектом із 24 автоматизованих тестів та експериментальною експлуатацією, під час якої опрацьовано 14 901 вимірювання й виявлено 346 аномалій.

# РОЗДІЛ 4. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

## 4.1. Характеристика умов праці при розробці та експлуатації програмного забезпечення

Розроблення програмного забезпечення для опрацювання даних з IoT-сенсорів передбачає тривалу роботу за персональним комп'ютером з монітором, клавіатурою та маніпулятором типу «миша». За класифікацією ДСН 3.3.6.042-99, виконувана робота належить до категорії Іа (легка фізична робота з витратами енергії до 139 Вт), проте супроводжується значним нервово-емоційним і зоровим напруженням, що характерно для операторського типу праці. [28]

Основне робоче обладнання: стаціонарний ПК або ноутбук з монітором з діагоналлю не менше 19–24 дюймів, клавіатура, маніпулятор, принтер, допоміжне IoT-обладнання (одноплатні мікрокомп'ютери, модулі сенсорів, MQTT-брокер на окремому пристрої) для тестування та розгортання системи. Робоче місце розташоване у приміщенні, призначеному для розміщення комп'ютерної техніки.

Відповідно до вимог НПАОП 0.00-7.15-18 (наказ Мінсоцполітики від 14.02.2018 № 207) роботодавець зобов'язаний забезпечити відповідність умов праці мінімальним вимогам безпеки й захисту здоров'я для всіх працівників, які використовують екранні пристрої незалежно від їх типу та моделі. Нижче розглянуто основні шкідливі та небезпечні виробничі чинники, що виникають на такому робочому місці, та заходи щодо їх нейтралізації. [26]

На робочому місці розробника програмного забезпечення діють такі основні шкідливі та небезпечні виробничі чинники:

- зорове і нервово-психічне напруження при тривалій роботі з екраном монітора;
- статичне м'язово-скелетне навантаження внаслідок незмінної пози під час сидячої роботи;

- електромагнітне випромінювання від монітора та периферійного обладнання;
- незадовільні параметри мікроклімату (температура, вологість, швидкість руху повітря);
- недостатня або нерівномірна освітленість робочої зони;
- підвищений рівень шуму від систем охолодження ПК та вентиляції приміщення;
- небезпека ураження електричним струмом від електроустановок;
- пожежна небезпека через займання ізоляції кабелів та електронних компонентів.

При розробленні програмного забезпечення додатково виникають специфічні чинники нервово-емоційного напруження: необхідність тривалої концентрації уваги при налагодженні алгоритмів обробки даних, аналізі журналів роботи системи в реальному часі, виявленні та усуненні помилок у логіці виявлення аномалій. Ці чинники підвищують ризик виникнення загального та зорового стомлення.

## 4.2. Ергономіка робочого місця розробника програмного забезпечення для пристроїв IoT

Параметри мікроклімату регламентуються ДСН 3.3.6.042-99 «Санітарні норми мікроклімату виробничих приміщень». Для робіт операторського типу, пов’язаних з нервово-емоційним напруженням у приміщеннях, де розміщена обчислювальна техніка, пунктом 1.1.4 цього документа встановлено обов’язкові оптимальні умови мікроклімату незалежно від пори року. Нормативні значення наведено в таблиці 4.1. [28]

Таблиця 4.1 — Оптимальні параметри мікроклімату для приміщень з ПК (ДСН 3.3.6.042-99)

| Параметр | Холодний період | Теплий період | Одиниця вимірювання |
|---|---|---|---|
| Температура повітря | 21–23 | 22–24 | °C |
| Відносна вологість | 40–60 | 40–60 | % |
| Швидкість руху повітря | ≤ 0,1 | ≤ 0,1 | м/с |

Для забезпечення нормативних параметрів мікроклімату в приміщенні передбачають: опалення з автоматичним регулюванням температури у холодний період; кондиціонування або примусову вентиляцію у теплий період; зволожувачі повітря за необхідності підтримання вологості в діапазоні 40–60 %. Теплове навантаження від ПК, сервера MQTT-брокера та додаткового IoT-обладнання враховують при проєктуванні системи кліматизації.

Освітлення нормується відповідно до ДБН В.2.5-28:2018 «Природне і штучне освітлення». Робота за монітором ПК при написанні коду, перегляді даних з IoT-сенсорів та аналізі логів відповідає IV розряду зорової роботи. Нормоване значення освітленості на горизонтальній робочій поверхні (площина столу) при системі комбінованого штучного освітлення становить 400 лк для загального освітлення і не менше 300 лк в площині екрана. [27]

Основні вимоги до організації освітлення:

- природне освітлення — бічне, вікна орієнтовані на північ або північний схід, що виключає пряме сонячне засвічення екрана;
- штучне освітлення — рівномірне загальне від люмінесцентних або світлодіодних світильників з розсіювачами, показник засліплення не перевищує 20;
- монітор розміщують перпендикулярно до площини вікон, щоб уникнути відблисків від скла;
- коефіцієнт пульсації освітленості — не більше 5 % (НПАОП 0.00-7.15-18, розділ III); [26]
- забороняється використання відкритих ламп розжарювання та галогенних ламп без плафонів-розсіювачів.

Організація робочого місця розробника регламентується НПАОП 0.00-7.15-18 (розділи II–IV). Мета ергономічного проєктування — забезпечити правильну поставу та мінімізувати статичне напруження м’язів, що особливо важливо при тривалому програмуванні та налагодженні системи обробки даних. [26]

Вимоги до робочої пози та розміщення обладнання:

- висота робочої поверхні столу — 680–800 мм відповідно до зросту; за необхідності регулюється підставкою для ніг;
- крісло — з регульованою висотою сидіння (400–500 мм), підлокітниками та поперековою підтримкою; спина відхилена на 100–110°;
- монітор — на відстані 600–700 мм від очей, верхній край екрана розташований на рівні очей або на 5–10 см нижче; кут нахилу — 10–15° від вертикалі;
- клавіатура та маніпулятор розміщуються так, щоб плечі були розслаблені, лікті зігнуті під кутом 90–100°;
- допоміжне IoT-обладнання (макетні плати, модулі давачів) розміщують у зоні досяжності на додатковому столику праворуч або ліворуч, поза основним полем зору.

Відповідно до пункту III.5 НПАОП 0.00-7.15-18, роботодавець зобов’язаний забезпечити проведення аналізу робочого місця та усунення виявлених ризиків. При роботі, пов’язаній з нервово-емоційним напруженням (розроблення та налагодження алгоритмів виявлення аномалій, аналіз потоків сенсорних даних), рекомендується встановлювати регламентовані перерви тривалістю 10–15 хвилин через кожні 45–60 хвилин безперервної роботи. [26]

### 4.3. Захист від електромагнітного випромінювання. Електро- та пожежобезпека.

Персональні комп’ютери, монітори та периферійне обладнання є джерелами електромагнітного випромінювання (ЕМВ) у широкому діапазоні

частот. НПАОП 0.00-7.15-18 (розділ V) встановлює, що екранні пристрої не повинні бути джерелом ризику для працівників: усе випромінювання, за винятком видимого діапазону електромагнітного спектра, має бути зведене до незначного рівня з погляду безпеки та охорони здоров'я. [26]

Сучасні монітори з рідкокристалічними (LCD) та органічними світлодіодними (OLED) панелями суттєво поступаються катодно-променевим трубкам (CRT) за рівнем ЕМВ і відповідають вимогам стандарту TCO Certified, що гарантує безпечні рівні низькочастотного електричного та магнітного полів. Для мінімізації впливу ЕМВ рекомендується:

- відстань від монітора до оператора — не менше 600 мм (з боку екрана) і не менше 1000 мм від бокових і задніх стінок сусідніх моніторів;
- застосовувати монітори зі сертифікатом ЕМВ-відповідності (TCO Certified, FCC, CE);
- не розміщувати IoT-шлюзи та роутери Wi-Fi безпосередньо на робочому столі на мінімальній відстані менше 300 мм від оператора;
- заземлювати корпуси ПК та периферійного обладнання.

Приміщення, де розміщений ПК і виконується розроблення та тестування програмного забезпечення для IoT-системи, класифікується як приміщення без підвищеної небезпеки (ПБЕ) відповідно до Правил улаштування електроустановок (ПУЕ): вологість повітря нормальна (відносна вологість до 75 %), температура в межах норми, підлога неструмопровідна. Приміщення для обслуговування, ремонту та налагодження ПК та IoT-обладнання належить за пожежовибухобезпекою до категорії В (НАПБ Б.03.002-2007) та до класу П-IIа (ПБЕ).

Основні вимоги електробезпеки при роботі з апаратурою розумного середовища:

- живлення ПК і допоміжного обладнання здійснюється від мережі 220/380 В, 50 Гц з обов'язковим захисним заземленням (зануленням) корпусів;
- до роботи з електроустановками допускаються особи, які пройшли інструктаж з електробезпеки та мають групу з електробезпеки не нижче I

(для загального персоналу) або II (для персоналу, який виконує під'єднання обладнання);

- забороняється залишати без нагляду ввімкнені у мережу пристрої з ознаками несправності (запах гарілого, нетипові звуки, іскріння);
- дроти та кабелі IoT-давачів (низьковольтні, 3,3–5 В, живлення від USB або PoE) прокладаються окремо від силових кабелів;
- розетки та подовжувачі не повинні бути перевантажені; сумарна потужність підключеного обладнання на одній розетковій групі — не більше 2000 Вт;
- після завершення роботи ПК та все допоміжне обладнання вимикається з мережі.

Пожежну безпеку в приміщеннях з комп'ютерною технікою регламентують ДБН В.1.1-7:2016 «Пожежна безпека об'єктів будівництва» та НАПБ А.01.001-2014 «Правила пожежної безпеки в Україні». Основними причинами пожеж у приміщеннях з ПК є: займання ізоляції електричних кабелів внаслідок перегріву або коротких замикань; займання запилених плат та вентиляторів блоків живлення; займання паперу та пластикових корпусів від несправного обладнання.

Вимоги до пожежної безпеки приміщення розробника:

- приміщення обладнане автоматичною пожежною сигналізацією (НАПБ Б.06.004-97) з адресними димовими сповіщувачами;
- первинні засоби пожежогасіння: вуглекислотний вогнегасник ВВК-2 або ВВК-3 (вміст вуглекислого газу не пошкоджує електронне обладнання); пісок; азбестова ковдра; заборонено використовувати порошкові або пінні вогнегасники поблизу увімкненого обладнання;
- усі виходи з приміщення позначені відповідними знаками та залишаються вільними;
- горючі матеріали (папір, картон, пластикові коробки) не зберігаються поблизу блоків живлення та обігрівальних приладів;

- системний блок ПК, сервер IoT-брокера та мережеве обладнання (роутер, комутатор) встановлені у стійках або на стелажах з вентиляційними зазорами не менше 100 мм з усіх боків.

Щороку проводяться протипожежні інструктажі, у яких фіксується порядок дій у разі пожежі: виклик служби 101, знеструмлення обладнання, евакуація персоналу відповідно до плану евакуації.

### 4.4. Заходи з охорони праці при розгортанні IoT-обладнання

Тривала робота за ПК при розробленні системи обробки IoT-даних пов’язана з підвищеним ризиком виникнення синдрому комп’ютерного зору (астенопія), м’язово-скелетних розладів (синдром карпального каналу, тендиніт, болі в шиї та спині), хронічного стомлення та нервово-психічного виснаження. Для профілактики цих станів рекомендується виконувати комплекс організаційних і медико-профілактичних заходів.

Режим праці та відпочинку:

- тривалість безперервної роботи за монітором — не більше 2 годин; після цього обов’язкова перерва тривалістю не менше 15 хвилин;
- регламентовані мікропаузи — 5 хвилин через кожні 45–60 хвилин роботи, під час яких рекомендовано виконувати вправи для очей і динамічну розминку;
- загальна тривалість роботи за монітором не повинна перевищувати 6 годин на зміну при 8-годинному робочому дні.

Гімнастика для очей (профілактика астенопії): кожні 20 хвилин перевести погляд на предмет, розташований на відстані не менше 6 метрів, і зосередитися на ньому протягом 20 секунд (правило «20-20-20»); виконати кругові рухи очима, кліпати протягом 10 секунд.

Профілактика м’язово-скелетних розладів: виконувати вправи для розтягнення м’язів шиї, плечей і зап’ясть під час мікропауз; чергувати роботу з клавіатурою з іншими видами діяльності (читання документації, наради); за

наявності передумов — використовувати ергономічну клавіатуру та вертикальну мишу.

Медичне забезпечення: попередній (при прийомі на роботу) і щорічні (для осіб, які працюють за ПК більше 50 % робочого часу) обов'язкові медичні огляди відповідно до наказу МОЗ України № 246 від 21.05.2007. Обстеження включає офтальмологічну перевірку, оцінку стану опорно-рухового апарату та нервової системи.

Специфічним аспектом роботи над проєктом є необхідність фізичного монтажу й тестування апаратної частини розумного середовища: підключення давачів температури, вологості, $CO_2$, освітленості та руху, налаштування MQTT-брокера на одноплатному мікрокомп'ютері (наприклад, Raspberry Pi), прокладання кабелів. Ці роботи потребують дотримання додаткових вимог охорони праці.

При монтажі та тестуванні IoT-давачів:

- низьковольтне обладнання (давачі 3,3–5 В, живлення USB) є безпечним за напругою, однак не захищене від статичної електрики; компоненти зберігають і монтують в антистатичних умовах (антистатичний браслет, антистатичний килимок);
- перед підключенням модулів до одноплатного комп'ютера переконатися у відповідності рівнів напруги (3,3 В або 5 В) та наявності необхідних резисторів захисту;
- мережевий роутер і MQTT-брокер (Mosquitto) підключають до мережі 220 В лише через заземлений подовжувач із захистом від перевантаження;
- при прокладанні кабелів давачів у приміщенні слід уникати механічних ушкоджень ізоляції (прокладка під килимком, у кабельних каналах); не допускати перетинання низьковольтних кабелів із мережами 220В;
- після завершення тестування обладнання переводять у штатний режим або знеструмлюють; не залишати безнаглядно пристрої у нештатному стані (перегрів, іскріння).

## 4.5. Висновки до четвертого розділу

У розділі проаналізовано умови праці при розробленні та експлуатації програмного забезпечення для опрацювання даних з IoT-сенсорів у розумному середовищі. Визначено основні шкідливі та небезпечні виробничі чинники: зорове й нервово-психічне навантаження, електромагнітне випромінювання, незадовільний мікроклімат, недостатня освітленість, небезпека ураження електричним струмом і пожежна небезпека.

На підставі чинних нормативних документів — НПАОП 0.00-7.15-18, ДСН 3.3.6.042-99, ДБН В.2.5-28:2018, ПУЕ та НАПБ А.01.001-2014 — сформульовано вимоги до параметрів мікроклімату, освітленості, ергономіки робочого місця, захисту від ЕМВ, електробезпеки та пожежної безпеки. Визначено заходи профілактики виробничо-обумовлених захворювань: режим праці й відпочинку, гімнастика для очей, динамічні перерви, медичні огляди. Окремо розглянуто безпеку при монтажі та тестуванні апаратної частини IoT-системи. Дотримання зазначених заходів забезпечує безпечні та здорові умови праці розробника протягом усього терміну виконання роботи. [26, 27, 28]

# ВИСНОВКИ

У кваліфікаційній роботі розв’язано актуальну задачу розроблення програмного забезпечення для опрацювання даних з IoT-сенсорів у розумному середовищі. Основні результати роботи такі:

1) проаналізовано предметну область: концепцію Інтернету речей і розумного середовища, рівневу архітектуру IoT-систем, протоколи передавання даних (передусім MQTT і HTTP) та характерні проблеми опрацювання сенсорних потоків;
2) спроєктовано трирівневу модульну архітектуру (ядро обробки, сховище, сервіс подання) та конвеєр опрацювання: валідація й нормалізація → виявлення аномалій → агрегація → збереження;
3) обґрунтовано й реалізовано три взаємодоповнювальні методи виявлення аномалій — контроль діапазонів, ковзний z-показник і контроль швидкості зміни — з прозорим поясненням кожного спрацювання;
4) реалізовано програмне забезпечення мовою Python із застосуванням FastAPI, прийманням даних через MQTT, вбудованим симулятором сенсорів, збереженням у SQLite та вебпанеллю моніторингу;
5) коректність роботи підтверджено комплектом із 24 автоматизованих тестів та експериментальною експлуатацією, під час якої опрацьовано 14 901 вимірювання й виявлено 346 аномалій.

Розроблене програмне забезпечення є модульним, прозорим, відтворюваним і невибагливим до ресурсів, що робить його придатним як для практичного застосування у системах моніторингу розумного житла, так і як основу для подальшого розвитку.

**Напрями подальшого розвитку.** Перспективними є: розширення набору методів виявлення аномалій за рахунок методів машинного навчання; додавання сценаріїв автоматизації та сповіщень; підтримка розподіленого розгортання та горизонтального масштабування; інтеграція із зовнішніми платформами візуалізації.

# ПЕРЕЛІК ДЖЕРЕЛ


1. Ashton K. That ‘Internet of Things’ Thing // RFID Journal. — 2009. — URL: https://www.rfidjournal.com/that-internet-of-things-thing.
2. Atzori L., Iera A., Morabito G. The Internet of Things: A survey // Computer Networks. — 2010. — Vol. 54, No. 15. — P. 2787–2805.
3. Gubbi J., Buyya R., Marusic S., Palaniswami M. Internet of Things (IoT): A vision, architectural elements, and future directions // Future Generation Computer Systems. — 2013. — Vol. 29, No. 7. — P. 1645–1660.
4. Al-Fuqaha A. et al. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications // IEEE Communications Surveys & Tutorials. — 2015. — Vol. 17, No. 4. — P. 2347–2376.
5. OASIS. MQTT Version 5.0. OASIS Standard. — 2019. — URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.
6. Eclipse Foundation. Eclipse Paho — MQTT and MQTT-SN client library. — URL: https://www.eclipse.org/paho/.
7. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures: PhD dissertation. — University of California, Irvine, 2000.
8. FastAPI Documentation. — URL: https://fastapi.tiangolo.com/.
9. Pydantic Documentation. — URL: https://docs.pydantic.dev/.
10. SQLite Documentation. — URL: https://www.sqlite.org/docs.html.
11. Uvicorn — ASGI web server for Python. — URL: https://www.uvicorn.org/.
12. Chandola V., Banerjee A., Kumar V. Anomaly Detection: A Survey // ACM Computing Surveys. — 2009. — Vol. 41, No. 3. — P. 1–58.
13. Hodge V. J., Austin J. A Survey of Outlier Detection Methodologies // Artificial Intelligence Review. — 2004. — Vol. 22. — P. 85–126.
14. Aggarwal C. C. Outlier Analysis. — 2nd ed. — Springer, 2017. — 466 p.
15. ISO 7730:2005. Ergonomics of the thermal environment — Analytical determination and interpretation of thermal comfort. — ISO, 2005.
16. ASHRAE Standard 62.1. Ventilation for Acceptable Indoor Air Quality. — ASHRAE, 2019.

17. Shi W., Cao J., Zhang Q., Li Y., Xu L. Edge Computing: Vision and Challenges // IEEE Internet of Things Journal. — 2016. — Vol. 3, No. 5. — P. 637–646.
18. Bormann C., Castellani A. P., Shelby Z. CoAP: An Application Protocol for Billions of Tiny Internet Nodes // IEEE Internet Computing. — 2012. — Vol. 16, No. 2. — P. 62–67.
19. van Rossum G., Drake F. L. Python 3 Reference Manual. — CreateSpace, 2009.
20. pytest Documentation. — URL: https://docs.pytest.org/.
21. Home Assistant. Open source home automation. — URL: https://www.home-assistant.io/.
22. ThingsBoard. Open-source IoT Platform. — URL: https://thingsboard.io/.
23. Node-RED. Low-code programming for event-driven applications. — URL: https://nodered.org/.
24. Domoticz. Home Automation System. — URL: https://www.domoticz.com/.
25. ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура та правила оформлювання. — Київ : ДП «УкрНДНЦ», 2016.
26. НПАОП 0.00-7.15-18. Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями. — Київ, 2018.
27. ДБН В.2.5-28:2018. Природне і штучне освітлення. — Київ : Мінрегіон України, 2018.
28. ДСН 3.3.6.042-99. Санітарні норми мікроклімату виробничих приміщень. — Київ, 1999.
29. Bishop C. M. Pattern Recognition and Machine Learning. — Springer, 2006. — 738 p.
30. Kreps J. I Heart Logs: Event Data, Stream Processing, and Data Integration. — O’Reilly Media, 2014.
31. Newman S. Building Microservices: Designing Fine-Grained Systems. — 2nd ed. — O’Reilly Media, 2021.

# ДОДАТОК А. ІНСТРУКЦІЯ З РОЗГОРТАННЯ ТА ЗАПУСКУ

Для розгортання системи потрібні Python версії 3.12 або новішої та менеджер пакетів pip. Послідовність дій:

```
# 1. Створення віртуального середовища та встановлення залежностей
python -m venv .venv
source .venv/bin/activate      # Windows: .venv\Scripts\activate
pip install -r requirements.txt

# 2. Наповнення бази демонстраційними даними (симуляція 24 год)
python scripts/seed_demo.py --hours 24 --step 60

# 3. Запуск REST-сервісу та вебпанелі
python scripts/run_api.py --port 8000
#    вебпанель:  http://127.0.0.1:8000/
#    документація API:  http://127.0.0.1:8000/docs

# 4. (опційно) подавання даних у реальному часі
python scripts/run_simulator.py --rate 1

# 5. Запуск автоматизованих тестів
pytest -q
```

Для приймання даних від реального брокера MQTT у файлі конфігурації config.yaml встановлюють параметр mqtt.enabled значення true та зазначають адресу брокера; повідомлення публікуються у тему вигляду smartenv/<кімната>/<давач>.

# ДОДАТОК Б. ЛІСТИНГ ПРОГРАМНОГО КОДУ

У додатку наведено повний лістинг основних модулів програмного забезпечення (пакет iot_processing). Код подано мовою Python без скорочень.

*Лістинг Б.1 — Модуль models.py (моделі даних і типи показників)*

```
"""
Data models for the IoT smart-environment processing system.

Defines the canonical schema for raw sensor messages,
validated/normalized
readings, anomaly events and aggregated statistics. Pydantic is
used so that
every message crossing a boundary (MQTT, REST API, storage) is
validated
against a single source of truth.
"""
from __future__ import annotations

from datetime import datetime, timezone
from enum import Enum
from typing import Optional

from pydantic import BaseModel, Field, field_validator


def utcnow() -> datetime:
    """Timezone-aware UTC timestamp (avoids naive-datetime
ambiguity)."""
    return datetime.now(timezone.utc)


class SensorType(str, Enum):
    """Physical quantity measured by a sensor in the smart
environment."""
    TEMPERATURE = "temperature"   # deg C
    HUMIDITY = "humidity"         # % relative humidity
    CO2 = "co2"                   # ppm
    LIGHT = "light"               # lux
    MOTION = "motion"             # 0/1 (presence)
    POWER = "power"               # W (instantaneous power draw)
    PRESSURE = "pressure"         # hPa


# Physically plausible operating ranges per sensor type. Readings
outside the
# hard range are rejected as invalid; readings inside the range
but outside the
# "comfort"/expected band are candidates for anomaly detection.
SENSOR_RANGES: dict[SensorType, tuple[float, float]] = {
    SensorType.TEMPERATURE: (-40.0, 85.0),
    SensorType.HUMIDITY: (0.0, 100.0),
```

```
    SensorType.CO2: (300.0, 10000.0),
    SensorType.LIGHT: (0.0, 100000.0),
    SensorType.MOTION: (0.0, 1.0),
    SensorType.POWER: (0.0, 5000.0),
    SensorType.PRESSURE: (800.0, 1200.0),
}

SENSOR_UNITS: dict[SensorType, str] = {
    SensorType.TEMPERATURE: "C",
    SensorType.HUMIDITY: "%",
    SensorType.CO2: "ppm",
    SensorType.LIGHT: "lux",
    SensorType.MOTION: "bool",
    SensorType.POWER: "W",
    SensorType.PRESSURE: "hPa",
}


class RawMessage(BaseModel):
    """A message exactly as published by a sensor / gateway.

    Sensors in the wild are messy: timestamps may be missing,
values may arrive
    as strings, sensor_type casing may vary. This model is
intentionally
    permissive; the pipeline's validation stage is what enforces
correctness.
    """
    sensor_id: str = Field(..., min_length=1, max_length=64)
    sensor_type: str
    value: float
    unit: Optional[str] = None
    room: Optional[str] = None
    ts: Optional[datetime] = None

    @field_validator("value", mode="before")
    @classmethod
    def coerce_value(cls, v):
        # Sensors / brokers sometimes deliver numbers as strings.
        if isinstance(v, str):
            return float(v.strip())
        return v


class Reading(BaseModel):
    """A validated, normalized reading ready to be stored and
analyzed."""
    sensor_id: str
    sensor_type: SensorType
    value: float
    unit: str
    room: str = "unknown"
    ts: datetime
```

```
    quality: float = Field(1.0, ge=0.0, le=1.0)  # data-quality
score [0..1]

    def to_row(self) -> tuple:
        return (
            self.sensor_id,
            self.sensor_type.value,
            self.value,
            self.unit,
            self.room,
            self.ts.isoformat(),
            self.quality,
        )


class AnomalySeverity(str, Enum):
    INFO = "info"
    WARNING = "warning"
    CRITICAL = "critical"


class Anomaly(BaseModel):
    """An anomaly / alert raised by the detection stage."""
    sensor_id: str
    sensor_type: SensorType
    room: str
    value: float
    ts: datetime
    severity: AnomalySeverity
    method: str            # which detector fired (e.g. "range",
"zscore", "rate")
    message: str


class AggregateWindow(BaseModel):
    """Aggregated statistics for one sensor over a time window."""
    sensor_id: str
    sensor_type: SensorType
    room: str
    window_start: datetime
    window_end: datetime
    count: int
    mean: float
    minimum: float
    maximum: float
    stdev: float
```

*Лістинг Б.2 — Модуль config.py (конфігурація)*

```
"""
Configuration management.

Loads settings from a YAML file (config.yaml) with environment-
variable
```

```
overrides, falling back to safe defaults so the system runs out-
of-the-box
with no configuration at all.
"""
from __future__ import annotations

import os
from dataclasses import dataclass, field, asdict
from typing import Any

import yaml


@dataclass
class MQTTConfig:
    enabled: bool = False
    host: str = "127.0.0.1"
    port: int = 1883
    topic: str = "smartenv/+/+"   # smartenv/<room>/<sensor_id>
    client_id: str = "iot-processor"
    keepalive: int = 60


@dataclass
class AnomalyConfig:
    zscore_threshold: float = 3.0      # std-devs from rolling
mean
    rolling_window: int = 50           # samples kept per sensor
for stats
    min_samples_for_zscore: int = 10   # warm-up before z-score is
trusted
    rate_limit_per_sensor: float = 100.0  # max plausible change
per second


@dataclass
class AggregationConfig:
    window_seconds: int = 60           # aggregation bucket size


@dataclass
class AppConfig:
    db_path: str = "data/readings.db"
    mqtt: MQTTConfig = field(default_factory=MQTTConfig)
    anomaly: AnomalyConfig = field(default_factory=AnomalyConfig)
    aggregation: AggregationConfig =
field(default_factory=AggregationConfig)

    def to_dict(self) -> dict[str, Any]:
        return asdict(self)


def load_config(path: str = "config.yaml") -> AppConfig:
```

```
    """Load configuration from YAML, applying defaults for any
missing keys."""
    data: dict[str, Any] = {}
    if os.path.exists(path):
        with open(path, "r", encoding="utf-8") as fh:
            data = yaml.safe_load(fh) or {}

    cfg = AppConfig()
    cfg.db_path = data.get("db_path", cfg.db_path)

    if "mqtt" in data:
        cfg.mqtt = MQTTConfig(**{**asdict(cfg.mqtt),
**data["mqtt"]})
    if "anomaly" in data:
        cfg.anomaly = AnomalyConfig(**{**asdict(cfg.anomaly),
**data["anomaly"]})
    if "aggregation" in data:
        cfg.aggregation = AggregationConfig(
            **{**asdict(cfg.aggregation), **data["aggregation"]}
        )

    # Environment overrides (useful for containerized deployment).
    cfg.db_path = os.environ.get("IOT_DB_PATH", cfg.db_path)
    if os.environ.get("IOT_MQTT_HOST"):
        cfg.mqtt.host = os.environ["IOT_MQTT_HOST"]
        cfg.mqtt.enabled = True

    return cfg
```

*Лістинг Б.3 — Модуль validation.py (валідація та нормалізація)*

```
"""
Validation and normalization stage.

Turns a permissive RawMessage into a strict Reading, or rejects
it. This is the
first defensive layer of the pipeline: garbage from a flaky sensor
must never
reach storage or analytics unflagged.
"""
from __future__ import annotations

from typing import Optional

from .models import (
    RawMessage,
    Reading,
    SensorType,
    SENSOR_RANGES,
    SENSOR_UNITS,
    utcnow,
)


class ValidationError(Exception):
```

```
    """Raised when a raw message cannot be turned into a valid
reading."""


def _normalize_unit(sensor_type: SensorType, value: float, unit:
Optional[str]) -> tuple[float, str]:
    """Convert common unit variants to the canonical unit for the
sensor type.

    Handles the most frequent real-world inconsistencies:
Fahrenheit
    temperatures and kilowatt power values.
    """
    canonical = SENSOR_UNITS[sensor_type]
    if unit is None:
        return value, canonical

    u = unit.strip().lower()

    if sensor_type is SensorType.TEMPERATURE and u in {"f",
"fahrenheit", "°f"}:
        return (value - 32.0) * 5.0 / 9.0, canonical
    if sensor_type is SensorType.POWER and u in {"kw",
"kilowatt"}:
        return value * 1000.0, canonical

    return value, canonical


def validate(raw: RawMessage) -> Reading:
    """Validate and normalize a raw message into a Reading.

    Raises ValidationError on hard failures (unknown type, out-of-
range value).
    Assigns a data-quality score in [0, 1]: missing timestamp or
missing unit
    each reduce quality slightly, since such readings are less
trustworthy.
    """
    # Resolve sensor type (case-insensitive).
    try:
        stype = SensorType(raw.sensor_type.strip().lower())
    except ValueError as exc:
        raise ValidationError(f"unknown sensor_type:
{raw.sensor_type!r}") from exc

    quality = 1.0

    value, unit = _normalize_unit(stype, raw.value, raw.unit)
    if raw.unit is None:
        quality -= 0.1  # we assumed the canonical unit

    lo, hi = SENSOR_RANGES[stype]
```

```
    if not (lo <= value <= hi):
        raise ValidationError(
            f"value {value} out of physical range [{lo}, {hi}] for
{stype.value}"
        )

    # Motion is binary: snap to 0/1.
    if stype is SensorType.MOTION:
        value = 1.0 if value >= 0.5 else 0.0

    ts = raw.ts
    if ts is None:
        ts = utcnow()
        quality -= 0.1  # ingestion time used as a proxy for event
time

    room = (raw.room or "unknown").strip().lower() or "unknown"

    return Reading(
        sensor_id=raw.sensor_id.strip(),
        sensor_type=stype,
        value=round(value, 4),
        unit=unit,
        room=room,
        ts=ts,
        quality=max(0.0, round(quality, 2)),
    )
```

*Лістинг Б.4 — Модуль anomaly.py (виявлення аномалій)*

```
"""
Anomaly detection stage.

Three complementary detectors run on each validated reading:

1. Range / comfort-band detector  - domain thresholds (e.g. CO2 >
1000 ppm is
   unhealthy) independent of history.
2. Rolling z-score detector       - flags statistical outliers
relative to a
   per-sensor rolling window (adapts to each sensor's normal
behaviour).
3. Rate-of-change detector        - flags physically implausible
jumps between
   consecutive readings (typical of a failing/spiking sensor).

State (rolling windows, last reading) is kept in-memory per
sensor. The design
is intentionally simple and explainable: every alert names the
detector that
fired and the value that triggered it, which matters for a
defensible system.
"""
from __future__ import annotations
```

```
import statistics
from collections import defaultdict, deque
from typing import Optional

from .config import AnomalyConfig
from .models import Anomaly, AnomalySeverity, Reading, SensorType


# Comfort / health bands used by the range detector. (low, high,
severity).
COMFORT_BANDS: dict[SensorType, tuple[float, float,
AnomalySeverity]] = {
    SensorType.TEMPERATURE: (16.0, 28.0, AnomalySeverity.WARNING),
    SensorType.HUMIDITY: (30.0, 65.0, AnomalySeverity.WARNING),
    SensorType.CO2: (0.0, 1000.0, AnomalySeverity.WARNING),
    SensorType.PRESSURE: (980.0, 1040.0, AnomalySeverity.INFO),
}

# CO2 above this is escalated to CRITICAL (drowsiness / poor air
quality).
CRITICAL_CO2_PPM = 2000.0


class AnomalyDetector:
    def __init__(self, cfg: AnomalyConfig):
        self.cfg = cfg
        self._windows: dict[str, deque[float]] = defaultdict(
            lambda: deque(maxlen=cfg.rolling_window)
        )
        self._last: dict[str, Reading] = {}

    def _key(self, r: Reading) -> str:
        return f"{r.sensor_id}:{r.sensor_type.value}"

    def detect(self, r: Reading) -> list[Anomaly]:
        """Run all detectors on a reading and return any anomalies
raised."""
        anomalies: list[Anomaly] = []
        anomalies.extend(self._range_check(r))
        z = self._zscore_check(r)
        if z is not None:
            anomalies.append(z)
        rate = self._rate_check(r)
        if rate is not None:
            anomalies.append(rate)

        # Update state AFTER detection so z-score reflects
history, not the
        # current point.
        self._windows[self._key(r)].append(r.value)
        self._last[self._key(r)] = r
        return anomalies
```

```
    # --- individual detectors ---------------------------------
-------------

    def _range_check(self, r: Reading) -> list[Anomaly]:
        out: list[Anomaly] = []
        band = COMFORT_BANDS.get(r.sensor_type)
        if band is None:
            return out
        lo, hi, severity = band
        if r.value < lo or r.value > hi:
            sev = severity
            if r.sensor_type is SensorType.CO2 and r.value >=
CRITICAL_CO2_PPM:
                sev = AnomalySeverity.CRITICAL
            direction = "below" if r.value < lo else "above"
            out.append(
                Anomaly(
                    sensor_id=r.sensor_id,
                    sensor_type=r.sensor_type,
                    room=r.room,
                    value=r.value,
                    ts=r.ts,
                    severity=sev,
                    method="range",
                    message=(
                        f"{r.sensor_type.value} {r.value}{r.unit}
is {direction} "
                        f"comfort band [{lo}, {hi}]"
                    ),
                )
            )
        return out

    def _zscore_check(self, r: Reading) -> Optional[Anomaly]:
        window = self._windows[self._key(r)]
        if len(window) < self.cfg.min_samples_for_zscore:
            return None
        mean = statistics.fmean(window)
        sd = statistics.pstdev(window)
        if sd == 0:
            return None
        z = abs(r.value - mean) / sd
        if z >= self.cfg.zscore_threshold:
            return Anomaly(
                sensor_id=r.sensor_id,
                sensor_type=r.sensor_type,
                room=r.room,
                value=r.value,
                ts=r.ts,
                severity=AnomalySeverity.WARNING,
                method="zscore",
                message=(
```

```
                    f"value {r.value} is {z:.1f} std-devs from
rolling mean "
                    f"{mean:.2f} (window={len(window)})"
                ),
            )
        return None

    def _rate_check(self, r: Reading) -> Optional[Anomaly]:
        prev = self._last.get(self._key(r))
        if prev is None:
            return None
        dt = (r.ts - prev.ts).total_seconds()
        if dt <= 0:
            return None
        rate = abs(r.value - prev.value) / dt
        if rate > self.cfg.rate_limit_per_sensor:
            return Anomaly(
                sensor_id=r.sensor_id,
                sensor_type=r.sensor_type,
                room=r.room,
                value=r.value,
                ts=r.ts,
                severity=AnomalySeverity.WARNING,
                method="rate",
                message=(
                    f"change rate {rate:.1f}/s exceeds limit "
                    f"{self.cfg.rate_limit_per_sensor}/s"
                ),
            )
        return None
```

*Лістинг Б.5 — Модуль aggregation.py (агрегація за вікнами)*

```
"""
Aggregation stage.

Maintains streaming time-window aggregates
(count/mean/min/max/stdev) per
sensor without re-scanning storage. Aggregates downsample high-
frequency raw
streams into compact summaries suitable for dashboards and trend
analysis.
"""
from __future__ import annotations

import statistics
from collections import defaultdict
from datetime import datetime, timedelta, timezone

from .models import AggregateWindow, Reading


class WindowAggregator:
    def __init__(self, window_seconds: int = 60):
        self.window = timedelta(seconds=window_seconds)
```

```
        # key -> (window_start, [values], room, sensor_type)
        self._buckets: dict[str, dict] = defaultdict(dict)

    def _key(self, r: Reading) -> str:
        return f"{r.sensor_id}:{r.sensor_type.value}"

    def add(self, r: Reading) -> AggregateWindow | None:
        """Add a reading. Returns a completed AggregateWindow when
the current
        window for that sensor rolls over, otherwise None."""
        key = self._key(r)
        bucket = self._buckets[key]
        completed: AggregateWindow | None = None

        if not bucket:
            bucket.update(start=r.ts, values=[r.value],
room=r.room,
                          stype=r.sensor_type)
            return None

        if r.ts - bucket["start"] >= self.window:
            completed = self._finalize(r.sensor_id, key, bucket,
end=r.ts)
            bucket.clear()
            bucket.update(start=r.ts, values=[r.value],
room=r.room,
                          stype=r.sensor_type)
        else:
            bucket["values"].append(r.value)
            bucket["room"] = r.room
        return completed

    def _finalize(self, sensor_id, key, bucket, end) ->
AggregateWindow:
        vals = bucket["values"]
        return AggregateWindow(
            sensor_id=sensor_id,
            sensor_type=bucket["stype"],
            room=bucket["room"],
            window_start=bucket["start"],
            window_end=end,
            count=len(vals),
            mean=round(statistics.fmean(vals), 4),
            minimum=min(vals),
            maximum=max(vals),
            stdev=round(statistics.pstdev(vals), 4) if len(vals) >
1 else 0.0,
        )

    def flush(self) -> list[AggregateWindow]:
        """Finalize all open windows (e.g. on shutdown)."""
        now = datetime.now(timezone.utc)
        out = []
```

```
        for key, bucket in list(self._buckets.items()):
            if bucket and bucket["values"]:
                sensor_id = key.split(":", 1)[0]
                out.append(self._finalize(sensor_id, key, bucket,
end=now))
        self._buckets.clear()
        return out
```

*Лістинг Б.6 — Модуль storage.py (сховище SQLite)*

```
"""
Storage layer (SQLite).

A thin, dependency-free persistence layer. SQLite is chosen
deliberately: it is
embedded, requires no server, supports concurrent reads, and keeps
the whole
system "offline-friendly" with a low memory footprint -
appropriate for an edge
/ single-node smart-environment gateway.
"""
from __future__ import annotations

import os
import sqlite3
import threading
from datetime import datetime
from typing import Iterable, Optional

from .models import Anomaly, Reading, SensorType


SCHEMA = """
CREATE TABLE IF NOT EXISTS readings (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    sensor_id   TEXT NOT NULL,
    sensor_type TEXT NOT NULL,
    value       REAL NOT NULL,
    unit        TEXT NOT NULL,
    room        TEXT NOT NULL,
    ts          TEXT NOT NULL,
    quality     REAL NOT NULL
);
CREATE INDEX IF NOT EXISTS idx_readings_sensor ON
readings(sensor_id, ts);
CREATE INDEX IF NOT EXISTS idx_readings_room   ON readings(room,
ts);

CREATE TABLE IF NOT EXISTS anomalies (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    sensor_id   TEXT NOT NULL,
    sensor_type TEXT NOT NULL,
    room        TEXT NOT NULL,
    value       REAL NOT NULL,
    ts          TEXT NOT NULL,
```

```
    severity    TEXT NOT NULL,
    method      TEXT NOT NULL,
    message     TEXT NOT NULL
);
CREATE INDEX IF NOT EXISTS idx_anomalies_ts ON anomalies(ts);

CREATE TABLE IF NOT EXISTS sensors (
    sensor_id   TEXT PRIMARY KEY,
    sensor_type TEXT NOT NULL,
    room        TEXT NOT NULL,
    last_seen   TEXT NOT NULL
);
"""


class Storage:
    def __init__(self, db_path: str):
        self.db_path = db_path
        parent = os.path.dirname(db_path)
        if parent:
            os.makedirs(parent, exist_ok=True)
        # check_same_thread=False + a lock lets the API and
ingestion threads
        # share one connection safely.
        self._conn = sqlite3.connect(db_path,
check_same_thread=False)
        self._conn.row_factory = sqlite3.Row
        self._lock = threading.Lock()
        self._conn.executescript(SCHEMA)
        self._conn.commit()

    # --- writes ---------------------------------------------------
-------------

    def insert_reading(self, r: Reading) -> None:
        with self._lock:
            self._conn.execute(
                "INSERT INTO readings "
                "(sensor_id, sensor_type, value, unit, room, ts,
quality) "
                "VALUES (?, ?, ?, ?, ?, ?, ?)",
                r.to_row(),
            )
            self._conn.execute(
                "INSERT INTO sensors (sensor_id, sensor_type,
room, last_seen) "
                "VALUES (?, ?, ?, ?) "
                "ON CONFLICT(sensor_id) DO UPDATE SET "
                "last_seen=excluded.last_seen,
room=excluded.room",
                (r.sensor_id, r.sensor_type.value, r.room,
r.ts.isoformat()),
            )
```

```
            self._conn.commit()

    def insert_anomalies(self, anomalies: Iterable[Anomaly]) ->
None:
        rows = [
            (a.sensor_id, a.sensor_type.value, a.room, a.value,
             a.ts.isoformat(), a.severity.value, a.method,
a.message)
            for a in anomalies
        ]
        if not rows:
            return
        with self._lock:
            self._conn.executemany(
                "INSERT INTO anomalies "
                "(sensor_id, sensor_type, room, value, ts,
severity, method, message) "
                "VALUES (?, ?, ?, ?, ?, ?, ?, ?)",
                rows,
            )
            self._conn.commit()

    # --- reads ------------------------------------------------
-------------

    def recent_readings(
        self,
        sensor_id: Optional[str] = None,
        room: Optional[str] = None,
        sensor_type: Optional[str] = None,
        limit: int = 200,
    ) -> list[dict]:
        q = "SELECT * FROM readings WHERE 1=1"
        params: list = []
        if sensor_id:
            q += " AND sensor_id = ?"
            params.append(sensor_id)
        if room:
            q += " AND room = ?"
            params.append(room)
        if sensor_type:
            q += " AND sensor_type = ?"
            params.append(sensor_type)
        q += " ORDER BY ts DESC LIMIT ?"
        params.append(limit)
        with self._lock:
            rows = self._conn.execute(q, params).fetchall()
        return [dict(row) for row in rows]

    def recent_anomalies(self, limit: int = 100) -> list[dict]:
        with self._lock:
            rows = self._conn.execute(
```

```
                "SELECT * FROM anomalies ORDER BY ts DESC LIMIT
?", (limit,)
            ).fetchall()
        return [dict(row) for row in rows]

    def list_sensors(self) -> list[dict]:
        with self._lock:
            rows = self._conn.execute(
                "SELECT * FROM sensors ORDER BY room, sensor_id"
            ).fetchall()
        return [dict(row) for row in rows]

    def stats(self) -> dict:
        with self._lock:
            n_readings = self._conn.execute(
                "SELECT COUNT(*) AS c FROM readings"
            ).fetchone()["c"]
            n_anomalies = self._conn.execute(
                "SELECT COUNT(*) AS c FROM anomalies"
            ).fetchone()["c"]
            n_sensors = self._conn.execute(
                "SELECT COUNT(*) AS c FROM sensors"
            ).fetchone()["c"]
            by_type = self._conn.execute(
                "SELECT sensor_type, COUNT(*) AS c FROM readings
GROUP BY sensor_type"
            ).fetchall()
        return {
            "readings": n_readings,
            "anomalies": n_anomalies,
            "sensors": n_sensors,
            "readings_by_type": {row["sensor_type"]: row["c"] for
row in by_type},
        }

    def aggregate(
        self, sensor_id: str, sensor_type: str, since:
Optional[datetime] = None
    ) -> dict:
        q = (
            "SELECT COUNT(*) AS count, AVG(value) AS mean, "
            "MIN(value) AS minimum, MAX(value) AS maximum "
            "FROM readings WHERE sensor_id = ? AND sensor_type =
?"
        )
        params: list = [sensor_id, sensor_type]
        if since is not None:
            q += " AND ts >= ?"
            params.append(since.isoformat())
        with self._lock:
            row = self._conn.execute(q, params).fetchone()
        return dict(row) if row else {}
```

```
    def close(self) -> None:
        with self._lock:
            self._conn.close()
```

*Лістинг Б.7 — Модуль pipeline.py (оркестратор конвеєра)*

```
"""
Processing pipeline orchestrator.

Wires the stages together into a single entry point:

    RawMessage -> validate -> detect anomalies -> aggregate ->
persist

The pipeline is transport-agnostic: it does not care whether a
message arrived
over MQTT, from the simulator, or from an HTTP POST. Each call
returns a small
result object so callers (API, tests) can react to what happened.
"""
from __future__ import annotations

from dataclasses import dataclass, field

from .aggregation import WindowAggregator
from .anomaly import AnomalyDetector
from .config import AppConfig
from .models import Anomaly, RawMessage, Reading
from .storage import Storage
from .validation import ValidationError, validate


@dataclass
class PipelineResult:
    accepted: bool
    reading: Reading | None = None
    anomalies: list[Anomaly] = field(default_factory=list)
    error: str | None = None


class Pipeline:
    def __init__(self, config: AppConfig, storage: Storage | None
= None):
        self.config = config
        self.storage = storage or Storage(config.db_path)
        self.detector = AnomalyDetector(config.anomaly)
        self.aggregator =
WindowAggregator(config.aggregation.window_seconds)
        self._processed = 0
        self._rejected = 0

    def process(self, raw: RawMessage) -> PipelineResult:
        # 1. Validate / normalize.
        try:
            reading = validate(raw)
```

```
        except ValidationError as exc:
            self._rejected += 1
            return PipelineResult(accepted=False, error=str(exc))

        # 2. Detect anomalies (uses rolling per-sensor state).
        anomalies = self.detector.detect(reading)

        # 3. Update streaming aggregates (side-effect; persisted
lazily).
        self.aggregator.add(reading)

        # 4. Persist.
        self.storage.insert_reading(reading)
        if anomalies:
            self.storage.insert_anomalies(anomalies)

        self._processed += 1
        return PipelineResult(accepted=True, reading=reading,
anomalies=anomalies)

    def process_dict(self, payload: dict) -> PipelineResult:
        """Convenience wrapper: build a RawMessage from a dict and
process it."""
        try:
            raw = RawMessage(**payload)
        except Exception as exc:  # pydantic ValidationError, etc.
            self._rejected += 1
            return PipelineResult(accepted=False,
error=f"malformed payload: {exc}")
        return self.process(raw)

    @property
    def counters(self) -> dict[str, int]:
        return {"processed": self._processed, "rejected":
self._rejected}
```

*Лістинг Б.8 — Модуль simulator.py (симулятор сенсорів)*

```
"""
Sensor simulator.

Generates physically plausible synthetic data for a multi-room
smart
environment so the whole system can be demonstrated without real
hardware.
Each virtual sensor follows a daily pattern (e.g. temperature
drifts, CO2 rises
when a room is "occupied") plus noise. Anomalies can be injected
at a
configurable probability to exercise the detection stage.
"""
from __future__ import annotations

import math
import random
```

```
from dataclasses import dataclass
from datetime import datetime, timezone

from .models import RawMessage, SensorType


@dataclass
class VirtualSensor:
    sensor_id: str
    sensor_type: SensorType
    room: str
    baseline: float
    amplitude: float          # daily swing amplitude
    noise: float              # gaussian noise std-dev
    phase_hours: float = 0.0  # shift of the daily peak

    def sample(self, t: datetime) -> float:
        # Fraction of the day [0, 1).
        seconds = t.hour * 3600 + t.minute * 60 + t.second
        frac = seconds / 86400.0
        daily = math.sin(2 * math.pi * (frac - self.phase_hours /
24.0))
        value = self.baseline + self.amplitude * daily +
random.gauss(0, self.noise)

        if self.sensor_type is SensorType.MOTION:
            # Occupancy more likely during daytime.
            p = 0.6 if 7 <= t.hour <= 23 else 0.05
            return 1.0 if random.random() < p else 0.0
        if self.sensor_type is SensorType.CO2:
            # CO2 rises with (simulated) occupancy during the day.
            occ = 1.0 if 8 <= t.hour <= 22 else 0.2
            value = self.baseline + occ * self.amplitude * (0.5 +
0.5 * daily)
            value += random.gauss(0, self.noise)
        return value


def default_environment() -> list[VirtualSensor]:
    """A small but representative smart home: 3 rooms, 7 sensor
instances."""
    return [
        VirtualSensor("temp-living", SensorType.TEMPERATURE,
"living_room",
                      baseline=22.0, amplitude=2.5, noise=0.2,
phase_hours=15),
        VirtualSensor("hum-living", SensorType.HUMIDITY,
"living_room",
                      baseline=45.0, amplitude=8.0, noise=1.0,
phase_hours=15),
        VirtualSensor("co2-living", SensorType.CO2, "living_room",
                      baseline=500.0, amplitude=600.0,
noise=30.0),
```

```
        VirtualSensor("temp-bed", SensorType.TEMPERATURE,
"bedroom",
                      baseline=20.0, amplitude=1.5, noise=0.2,
phase_hours=15),
        VirtualSensor("motion-bed", SensorType.MOTION, "bedroom",
                      baseline=0, amplitude=0, noise=0),
        VirtualSensor("light-kitchen", SensorType.LIGHT,
"kitchen",
                      baseline=300.0, amplitude=250.0, noise=20.0,
phase_hours=15),
        VirtualSensor("power-kitchen", SensorType.POWER,
"kitchen",
                      baseline=150.0, amplitude=120.0, noise=15.0,
phase_hours=18),
    ]


class Simulator:
    def __init__(self, sensors: list[VirtualSensor] | None = None,
                 anomaly_prob: float = 0.01, seed: int | None =
None):
        self.sensors = sensors or default_environment()
        self.anomaly_prob = anomaly_prob
        if seed is not None:
            random.seed(seed)

    def tick(self, t: datetime | None = None) -> list[RawMessage]:
        """Produce one reading per sensor for timestamp t
(default: now)."""
        t = t or datetime.now(timezone.utc)
        out: list[RawMessage] = []
        for s in self.sensors:
            value = s.sample(t)
            # Occasionally inject a gross anomaly to exercise
detection.
            if random.random() < self.anomaly_prob and
s.sensor_type not in (
                SensorType.MOTION,
            ):
                value *= random.choice([2.5, 3.0, -1.0, 0.0]) or
3.0
            out.append(
                RawMessage(
                    sensor_id=s.sensor_id,
                    sensor_type=s.sensor_type.value,
                    value=round(value, 3),
                    room=s.room,
                    ts=t,
                )
            )
        return out
```

*Лістинг Б.9 — Модуль mqtt_ingest.py (приймання через MQTT)*

```
"""
```

```
MQTT ingestion adapter (optional).

Subscribes to a topic on an MQTT broker and feeds incoming JSON
payloads into
the processing pipeline. MQTT is the de-facto messaging protocol
for IoT
(lightweight publish/subscribe over TCP), so this is the
"production" ingestion
path; the simulator can also publish to the same broker.

This module degrades gracefully: if paho-mqtt is not installed or
no broker is
reachable, the rest of the system still runs (the simulator can
feed the
pipeline directly).

Expected topic layout:  smartenv/<room>/<sensor_id>
Expected payload (JSON): {"sensor_type": "...", "value": ...,
"unit": "...",
                          "ts": "ISO-8601 (optional)"}
"""
from __future__ import annotations

import json
import logging

from .pipeline import Pipeline

log = logging.getLogger("iot.mqtt")


class MQTTIngestor:
    def __init__(self, pipeline: Pipeline):
        self.pipeline = pipeline
        self.cfg = pipeline.config.mqtt
        self._client = None

    def _on_connect(self, client, userdata, flags, rc,
properties=None):
        log.info("connected to MQTT broker rc=%s; subscribing to
%s",
                 rc, self.cfg.topic)
        client.subscribe(self.cfg.topic)

    def _on_message(self, client, userdata, msg):
        try:
            payload = json.loads(msg.payload.decode("utf-8"))
        except (ValueError, UnicodeDecodeError) as exc:
            log.warning("dropping non-JSON message on %s: %s",
msg.topic, exc)
            return
```

```
        # Derive room / sensor_id from the topic if absent in the
payload.
        parts = msg.topic.split("/")
        if len(parts) >= 3:
            payload.setdefault("room", parts[1])
            payload.setdefault("sensor_id", parts[2])

        result = self.pipeline.process_dict(payload)
        if not result.accepted:
            log.warning("rejected message on %s: %s", msg.topic,
result.error)
        elif result.anomalies:
            for a in result.anomalies:
                log.info("ANOMALY [%s] %s", a.severity.value,
a.message)

    def start(self) -> None:
        """Connect and block in the MQTT network loop."""
        import paho.mqtt.client as mqtt  # imported lazily

        self._client = mqtt.Client(client_id=self.cfg.client_id)
        self._client.on_connect = self._on_connect
        self._client.on_message = self._on_message
        log.info("connecting to %s:%s", self.cfg.host,
self.cfg.port)
        self._client.connect(self.cfg.host, self.cfg.port,
self.cfg.keepalive)
        self._client.loop_forever()

    def stop(self) -> None:
        if self._client is not None:
            self._client.disconnect()
```

*Лістинг Б.10 — Модуль api.py (REST-інтерфейс і вебпанель)*

```
"""
REST API + dashboard host (FastAPI).

Exposes the processing system over HTTP:

  POST /ingest             - submit a single raw reading (JSON)
  POST /ingest/batch       - submit many readings at once
  GET  /sensors            - list known sensors and last-seen time
  GET  /readings           - query recent readings (filter by
sensor/room/type)
  GET  /anomalies          - recent anomalies / alerts
  GET  /stats              - system-wide counters
  GET  /aggregate          - aggregate stats for one sensor
  GET  /healthz            - liveness probe
  GET  /                   - interactive dashboard (static HTML)

The same Pipeline instance is shared across requests; SQLite + an
internal lock
make concurrent access safe.
"""
```

```
from __future__ import annotations

import os
from datetime import datetime, timedelta, timezone
from typing import Optional

from fastapi import FastAPI, HTTPException, Query
from fastapi.responses import FileResponse, JSONResponse
from pydantic import BaseModel

from .config import load_config
from .models import RawMessage
from .pipeline import Pipeline

STATIC_DIR = os.path.join(os.path.dirname(__file__), "..", "..",
"static")

config = load_config(os.environ.get("IOT_CONFIG", "config.yaml"))
pipeline = Pipeline(config)

app = FastAPI(
    title="IoT Smart-Environment Data Processing",
    version="1.0.0",
    description="Processing pipeline for IoT sensor data in a
smart environment.",
)


class IngestResponse(BaseModel):
    accepted: bool
    anomalies: int
    error: Optional[str] = None


@app.post("/ingest", response_model=IngestResponse)
def ingest(msg: RawMessage):
    result = pipeline.process(msg)
    return IngestResponse(
        accepted=result.accepted,
        anomalies=len(result.anomalies),
        error=result.error,
    )


@app.post("/ingest/batch")
def ingest_batch(msgs: list[RawMessage]):
    accepted = 0
    rejected = 0
    anomalies = 0
    for m in msgs:
        r = pipeline.process(m)
        if r.accepted:
            accepted += 1
```

```
            anomalies += len(r.anomalies)
        else:
            rejected += 1
    return {"accepted": accepted, "rejected": rejected,
"anomalies": anomalies}


@app.get("/sensors")
def sensors():
    return pipeline.storage.list_sensors()


@app.get("/readings")
def readings(
    sensor_id: Optional[str] = None,
    room: Optional[str] = None,
    sensor_type: Optional[str] = None,
    limit: int = Query(200, ge=1, le=5000),
):
    return pipeline.storage.recent_readings(
        sensor_id=sensor_id, room=room, sensor_type=sensor_type,
limit=limit
    )


@app.get("/anomalies")
def anomalies(limit: int = Query(100, ge=1, le=2000)):
    return pipeline.storage.recent_anomalies(limit=limit)


@app.get("/stats")
def stats():
    return {**pipeline.storage.stats(), **pipeline.counters}


@app.get("/aggregate")
def aggregate(
    sensor_id: str,
    sensor_type: str,
    minutes: int = Query(60, ge=1, le=10080),
):
    since = datetime.now(timezone.utc) -
timedelta(minutes=minutes)
    agg = pipeline.storage.aggregate(sensor_id, sensor_type,
since=since)
    if not agg or agg.get("count", 0) == 0:
        raise HTTPException(status_code=404, detail="no data for
sensor in window")
    return agg


@app.get("/healthz")
def healthz():
```

```
    return {"status": "ok", "version": app.version}


@app.get("/")
def dashboard():
    index = os.path.join(STATIC_DIR, "dashboard.html")
    if os.path.exists(index):
        return FileResponse(index)
    return JSONResponse({"message": "dashboard not found; API is
running"})
```